

CORE JAVA

Volume II: Advanced Features

FOURTEENTH EDITION

ORACLE

Cay S. Horstmann

Core Java

Volume II: Advanced Features

Fourteenth Edition

Cay S. Horstmann

.

Cover image: surangaw/stock.adobe.com
Figures 3.3, 4.8: Mozilla Foundation
Figures 4.1-4.5, 5.3, 5.4: Microsoft Corporation
Figure 4.6: USPS
Figures 5.6, 7.1-7.6, 9.7, 10.1, 10.3, 10.8, 10.10, 10.11, 10.14-10.16, 11.4, 11.5, 11.8-11.27, 11.29-11.34, 12.1-12.5, 12.7-12.10, 12.14-12.26, 12.28-12.30, 12.39, 12.44-12.46, 12.54-12.56, 12.60, 12.62: Oracle Corporation
Figure 9.3: HHD Software Ltd
Figure 12.51: Shao-Chun Wang/123RF
Figures 12.57-12.59: Pearson Education

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Please contact us with concerns about any potential bias at pearson.com/en-us/report-bias.html.

Author websites are not owned or managed by Pearson.

Visit us on the Web: informit.com

Library of Congress Control Number: 2025945104

Copyright © 2026 Pearson Education, Inc.
Hoboken, New Jersey

Portions copyright © 1996-2013 Oracle and/or its affiliates. All Rights Reserved.

Oracle America Inc. does not make any representations or warranties as to the accuracy, adequacy or completeness of any information contained in this work, and is not responsible for any errors or omissions.

The views expressed in this book are those of the author and do not necessarily reflect the views of Oracle.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services. The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft® Windows®, and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit pearson.com/en-us/global-permission-granting.html.

ISBN-13: 978-0-13-555869-0
ISBN-10: 0-13-555869-7

Contents

Preface

Acknowledgments

1. Streams

1.1. From Iterating to Stream Operations
1.2. Stream Creation
1.3. The filter, map, and flatMap Methods
1.4. Extracting Substreams and Combining Streams
1.5. Other Stream Transformations
1.6. Simple Reductions
1.7. The Optional Type

1.7.1. Getting an Optional Value
1.7.2. Consuming an Optional Value
1.7.3. Pipelining Optional Values
1.7.4. How Not to Work with Optional Values
1.7.5. Creating Optional Values
1.7.6. Composing Optional Value Functions with flatMap
1.7.7. Turning an Optional into a Stream

1.8. Collecting Results
1.9. Collectors

1.9.1. Collecting into Collections
1.9.2. Collecting into Maps
1.9.3. Grouping and Partitioning
1.9.4. Downstream Collectors
1.9.5. Implementing Collectors

1.10. Reduction Operations
1.11. Gatherers

1.11.1. Predefined Gatherers
1.11.2. Implementing Gatherers

1.12. Primitive Type Streams
1.13. Parallel Streams

2. Input and Output

2.1. Input/Output Streams

2.1.1. Reading and Writing Bytes
2.1.2. The Complete Stream Zoo
2.1.3. Combining Input/Output Stream Filters
2.1.4. Text Input and Output
2.1.5. How to Read Text Input
2.1.6. How to Write Text Output
2.1.7. Saving Objects in Text Format
2.1.8. Character Encodings
2.1.9. Default Character Encodings
2.1.10. Reading Character Input

2.2. Reading and Writing Binary Data

2.2.1. The DataInput and DataOutput Interfaces
2.2.2. Random-Access Files
2.2.3. ZIP Archives

2.3. Working with Files

2.3.1. Paths
2.3.2. Reading and Writing Files
2.3.3. Creating Files and Directories
2.3.4. Copying, Moving, and Deleting Files
2.3.5. Getting File Information
2.3.6. Visiting Directory Entries
2.3.7. Glob Patterns
2.3.8. Directory Streams

2.3.9. ZIP File Systems

2.4. Memory-Mapped Files

2.4.1. Memory-Mapped File Performance
2.4.2. The Buffer Data Structure

2.5. File Locking
2.6. Object Input/Output Streams and Serialization

2.6.1. Saving and Loading Serializable Objects
2.6.2. Understanding the Object Serialization File Format
2.6.3. Transient Fields
2.6.4. The readObject and writeObject Methods
2.6.5. The readExternal and writeExternal Methods
2.6.6. The readResolve and writeReplace Methods
2.6.7. Versioning
2.6.8. Deserialization and Security
2.6.9. Using Serialization for Cloning

2.7. Regular Expressions

2.7.1. The Regular Expression Syntax
2.7.2. Testing a Match
2.7.3. Finding All Matches in a String
2.7.4. Groups
2.7.5. Splitting along Delimiters
2.7.6. Replacing Matches
2.7.7. Flags

3. XML

3.1. Introducing XML
3.2. The Structure of an XML Document
3.3. Parsing an XML Document
3.4. Validating XML Documents

3.4.1. Document Type Definitions
3.4.2. XML Schema
3.4.3. A Practical Example

3.5. Locating Information with XPath
3.6. Using Namespaces
3.7. Streaming Parsers

3.7.1. Using the SAX Parser
3.7.2. Using the StAX Parser

3.8. Generating XML Documents

3.8.1. Documents without Namespaces
3.8.2. Documents with Namespaces
3.8.3. Writing Documents
3.8.4. Writing an XML Document with StAX

3.9. XSL Transformations

4. Networking

4.1. Connecting to a Server

4.1.1. Using Telnet
4.1.2. Connecting to a Server with Java
4.1.3. Socket Timeouts
4.1.4. Internet Addresses

4.2. Implementing Servers

4.2.1. Server Sockets
4.2.2. Serving Multiple Clients
4.2.3. Half-Close
4.2.4. Interruptible Sockets
4.2.5. Secure Socket Communication

4.3. Getting Web Data

4.3.1. URLs and URIs
4.3.2. Using a URLConnection to Retrieve Information

4.3.3. Posting Form Data

4.4. The HTTP Client

4.4.1. The HttpClient Class
4.4.2. The HttpRequest class and Body Publishers
4.4.3. The HttpResponse Interface and Body Handlers
4.4.4. Asynchronous Processing

4.5. The Simple HTTP Server

4.5.1. The Command-Line Tool
4.5.2. The HTTP Server API
4.5.3. Handlers
4.5.4. Filters

4.6. Sending E-Mail

5. Database Programming

5.1. The Design of JDBC

5.1.1. JDBC Driver Types
5.1.2. Typical Uses of JDBC

5.2. The Structured Query Language
5.3. JDBC Configuration

5.3.1. Database URLs
5.3.2. Driver JAR Files
5.3.3. Starting the Database
5.3.4. Connecting to the Database

5.4. Working with JDBC Statements

5.4.1. Executing SQL Statements
5.4.2. Managing Connections, Statements, and Result Sets
5.4.3. Analyzing SQL Exceptions
5.4.4. Populating a Database

5.5. Query Execution

5.5.1. Prepared Statements
5.5.2. Reading and Writing LOBs
5.5.3. SQL Escapes
5.5.4. Multiple Results
5.5.5. Retrieving Autogenerated Keys

5.6. Scrollable and Updatable Result Sets

5.6.1. Scrollable Result Sets
5.6.2. Updatable Result Sets

5.7. Row Sets

5.7.1. Constructing Row Sets
5.7.2. Cached Row Sets

5.8. Metadata
5.9. Transactions

5.9.1. Programming Transactions with JDBC
5.9.2. Save Points
5.9.3. Batch Updates
5.9.4. Advanced SQL Types

5.10. Connection Management in Web and Enterprise Applications

6. The Date and Time API

6.1. The Time Line
6.2. Local Dates
6.3. Date Adjusters
6.4. Local Time
6.5. Zoned Time
6.6. Formatting and Parsing
6.7. Interoperating with Legacy Code

7. Internationalization

7.1. Locales

7.1.1. Why Locales?
7.1.2. Specifying Locales
7.1.3. The Default Locale
7.1.4. Display Names

7.2. Number Formats

7.2.1. Formatting Numeric Values
7.2.2. The DecimalFormat Class
7.2.3. Currencies

7.3. Date and Time
7.4. Collation and Normalization
7.5. Message Formatting

7.5.1. Formatting Numbers and Dates
7.5.2. Choice Formats

7.6. Text Boundaries
7.7. Text Input and Output

7.7.1. Text Files
7.7.2. Line Endings
7.7.3. The Console
7.7.4. The UTF-8 Byte Order Mark
7.7.5. Character Encoding of Source Files

7.8. Resource Bundles

7.8.1. Locating Resource Bundles
7.8.2. Property Files
7.8.3. Bundle Classes

7.9. A Complete Example

8. Compiling and Scripting

8.1. The Compiler API

8.1.1. Invoking the Compiler
8.1.2. Launching a Compilation Task
8.1.3. Capturing Diagnostics

8.1.4. Reading Source Files from Memory
8.1.5. Writing Byte Codes to Memory
8.1.6. An Example: Dynamic Java Code Generation

8.2. Scripting for the Java Platform

8.2.1. Getting a Scripting Engine
8.2.2. Script Evaluation and Bindings
8.2.3. Redirecting Input and Output
8.2.4. Calling Scripting Functions and Methods
8.2.5. Compiling a Script
8.2.6. An Example: Script Sheets

9. Security

9.1. Class Loaders

9.1.1. The Class-Loading Process
9.1.2. The Class Loader Hierarchy
9.1.3. Using Class Loaders as Namespaces
9.1.4. Writing Your Own Class Loader
9.1.5. Bytecode Verification

9.2. User Authentication

9.2.1. The JAAS Framework
9.2.2. JAAS Login Modules

9.3. Digital Signatures

9.3.1. Message Digests
9.3.2. Password Hashing
9.3.3. Message Signing
9.3.4. Verifying a Signature
9.3.5. The Authentication Problem
9.3.6. Certificate Signing
9.3.7. Code Signing

9.4. Encryption

9.4.1. Symmetric Ciphers

9.4.2. Key Generation
9.4.3. Cipher Streams
9.4.4. Public Key Ciphers

10. Graphical User Interface Programming

10.1. A History of Java User Interface Toolkits
10.2. Displaying Frames

10.2.1. Creating a Frame
10.2.2. Frame Properties

10.3. Displaying Information in a Component

10.3.1. Working with 2D Shapes
10.3.2. Using Color
10.3.3. Using Fonts
10.3.4. Displaying Images

10.4. Event Handling

10.4.1. Basic Event Handling Concepts
10.4.2. Example: Handling a Button Click
10.4.3. Specifying Listeners Concisely
10.4.4. Adapter Classes
10.4.5. Actions
10.4.6. Mouse Events
10.4.7. The AWT Event Hierarchy

10.5. The Preferences API

11. User Interface Components with Swing

11.1. Swing and the Model-View-Controller Design Pattern
11.2. Introduction to Layout Management

11.2.1. Layout Managers
11.2.2. Border Layout
11.2.3. Grid Layout

11.3. Text Input

11.3.1. Text Fields
11.3.2. Labels and Labeling Components
11.3.3. Password Fields
11.3.4. Text Areas
11.3.5. Scroll Panes

11.4. Choice Components

11.4.1. Checkboxes
11.4.2. Radio Buttons
11.4.3. Borders
11.4.4. Combo Boxes
11.4.5. Sliders

11.5. Menus

11.5.1. Menu Building
11.5.2. Icons in Menu Items
11.5.3. Checkbox and Radio Button Menu Items
11.5.4. Pop-Up Menus
11.5.5. Keyboard Mnemonics and Accelerators
11.5.6. Enabling and Disabling Menu Items
11.5.7. Toolbars
11.5.8. Tooltips

11.6. The Grid Bag Layout

11.6.1. Grid Bag Basics
11.6.2. The gridx, gridy, gridwidth, and gridheight Parameters
11.6.3. Weight Fields
11.6.4. The fill and anchor Parameters
11.6.5. Padding
11.6.6. Alternative Method to Specify the gridx, gridy, gridwidth, and gridheight Parameters
11.6.7. A Grid Bag Layout Recipe
11.6.8. A Helper Class to Tame the Grid Bag Constraints

11.7. Custom Layout Managers
11.8. Dialog Boxes

11.8.1. Option Panes
11.8.2. Creating Dialogs
11.8.3. Data Exchange
11.8.4. File Dialogs

12. Advanced Swing and Graphics

12.1. Tables

12.1.1. A Simple Table
12.1.2. Table Models

12.2. Working with Rows and Columns

12.2.1. Column Classes
12.2.2. Accessing Table Columns
12.2.3. Resizing Columns
12.2.4. Resizing Rows
12.2.5. Selecting Rows, Columns, and Cells
12.2.6. Sorting Rows
12.2.7. Filtering Rows
12.2.8. Hiding and Displaying Columns

12.3. Cell Rendering and Editing

12.3.1. Rendering Cells
12.3.2. Rendering the Header
12.3.3. Editing Cells
12.3.4. Custom Editors

12.4. Trees

12.4.1. Simple Trees
12.4.2. Editing Trees and Tree Paths
12.4.3. Node Enumeration
12.4.4. Rendering Nodes
12.4.5. Listening to Tree Events
12.4.6. Custom Tree Models

12.5. Advanced AWT

12.5.1. The Rendering Pipeline
12.5.2. The Shape Class Hierarchy
12.5.3. Constructing Shapes
12.5.4. Areas
12.5.5. Strokes
12.5.6. Paint
12.5.7. Coordinate Transformations
12.5.8. Clipping
12.5.9. Transparency and Composition

12.6. Raster Images

12.6.1. Readers and Writers for Images
12.6.2. Obtaining Readers and Writers for Image File Types
12.6.3. Reading and Writing Files with Multiple Images
12.6.4. Image Manipulation
12.6.5. Filtering Images

12.7. Printing

12.7.1. Graphics Printing
12.7.2. Multiple-Page Printing
12.7.3. Print Services
12.7.4. Stream Print Services
12.7.5. Printing Attributes

13. The Foreign Functions and Memory API

13.1. Using JNI to Call C Code from a Java Program
13.2. Using FFM to Call a Foreign Function
13.3. Arenas
13.4. Memory Segments
13.5. Memory Layout
13.6. Looking Up and Invoking Foreign Functions
13.7. Callbacks
13.8. Advanced Topics

13.8.1. Accessing Memory With Variable Handles
13.8.2. Calling a Foreign Varargs Function
13.8.3. Capturing errno

Index

Preface

To the Reader

The book you have in your hands is the second volume of the thirteenth edition of Core Java, fully updated for Java 25. The first volume covers the essential features of the language; this volume deals with the advanced topics that a programmer needs to know for professional software development. Thus, as with the first volume and the previous editions of this book, we are still targeting programmers who want to put Java technology to work in real projects.

With the explosive growth of the Java class library, a one-volume treatment of all the features of Java that serious programmers need to know is simply not possible. Hence, the book is broken up into two volumes.

This first volume concentrates on the fundamental concepts of the Java language:

- Object-oriented programming
- Reflection and proxies
- Interfaces and inner classes
- Exception handling
- Generic programming
- The collections framework
- Concurrency
- Annotations
- The Java platform module system

This second volume goes further into the most important libraries. For twelve editions, user interface programming was considered fundamental, but the time has come to recognize

that it is no more, and to move it into the second volume. The volume includes detailed discussions of these topics:

- The Stream API
- File processing and regular expressions
- Databases
- XML processing
- Scripting and Compiling APIs
- Internationalization
- Network programming
- Graphical user interface design
- Graphics programming
- Foreign functions and memory

As is the case with any book, errors and inaccuracies are inevitable. Should you find any in this book, we would very much like to hear about them. Of course, we would prefer to hear about them only once. For this reason, we have put up a web site at https://horstmann.com/corejava with a FAQ, bug fixes, and workarounds. Strategically placed at the end of the bug report web page (to encourage you to read the previous reports) is a form that you can use to report bugs or problems and to send suggestions for improvements for future editions.

A Tour of This Book

The chapters in this book are, for the most part, independent of each other. You should be able to delve into whatever topic interests you the most and read the chapters in any order.

In Chapter 1, you will learn all about the Java stream library that brings a modern flavor to processing data, by specifying what you want without describing in detail how the result should be obtained. This allows the stream library to focus on an optimal evaluation strategy, which is particularly advantageous for executing computations in parallel with multiple cores.

The topic of Chapter 2 is input and output handling (I/O). In Java, all input and output is handled through input/output

streams. These streams (not to be confused with those in Chapter 1) let you deal, in a uniform manner, with communications among various sources of data, such as files, network connections, or memory blocks. We include detailed coverage of the reader and writer classes that make it easy to deal with Unicode. We show you what goes on under the hood when you use the object serialization mechanism, which makes saving and loading objects easy and convenient. We then move on to regular expressions and working with files and paths. Throughout this chapter, you will find welcome enhancements in recent Java versions.

Chapter 3 covers XML. We show you how to parse XML files, how to generate XML, and how to use XSL transformations. As a useful example, we show you how to specify the layout of a Swing form in XML. We also discuss the XPath API, which makes finding needles in XML haystacks much easier.

Chapter 4 covers the networking API. Java makes it phenomenally easy to do complex network programming. We show you how to make network connections to servers, how to implement your own servers, and how to make HTTP connections. This chapter includes coverage of the new HTTP client.

Chapter 5 covers database programming. The focus is on JDBC, the Java database connectivity API that lets Java programs connect to relational databases. We show you how to write useful programs to handle realistic database chores, using a core subset of the JDBC API. (A complete treatment of the JDBC API would require a book almost as big as this one.)

Java had two prior attempts at libraries for handling date and time. The third one was the charm in Java 8. In Chapter 6, you will learn how to deal with the complexities of calendars and time zones, using the new date and time library.

Chapter 7 discusses a feature that we believe can only grow in importance: internationalization. The Java programming language is one of the few languages designed from the start to handle Unicode, but the internationalization support on the

Java platform goes much further. As a result, you can internationalize Java applications so that they cross not only platforms but country boundaries as well. For example, we show you how to write a retirement calculator that uses either English, German, or Chinese languages.

Chapter 8 discusses the scripting and compiler APIs that allow your program to call code in scripting languages such as JavaScript or Groovy, and to compile Java code.

Chapter 9 takes up the Java security model, user authentication, and the cryptographic functions in the Java security library. You will learn about important features such as message and code signing, authorization and authentication, and encryption. We conclude with examples that use the AES and RSA encryption algorithms.

Chapter 10 provides an introduction into GUI programming. I show how you can make windows, how to paint on them, how to draw with geometric shapes, how to format text in multiple fonts, and how to display images. Next, you'll see how to write code that responds to events, such as mouse clicks or key presses.

Chapter 11 discusses the Swing GUI toolkit in great detail. The Swing toolkit allows you to build cross-platform graphical user interfaces. You'll learn all about the various kinds of buttons, text components, borders, sliders, list boxes, menus, and dialog boxes.

Chapter 12 contains advanced Swing material, especially the important but complex tree and table components. We also cover the Java 2D API, which you can use to create realistic drawings and special effects. Of course, not many programmers need to program Swing user interfaces these days, so we pay particular attention to features that are useful for images that can be generated on a server.

Chapter 13 takes up foreign functions, which let you call code written in other programming languages. At times, you need to turn to the operating system's API for your target platform

when you interact with a device or service that is not supported by Java. Or you may want to call a battle-tested library for machine learning or data analysis.

As always, all chapters have been completely revised for the latest version of Java. Outdated material has been removed, and the new APIs up to Java 25 are covered in detail.

Conventions

As is common in many computer books, I use monospace type to represent computer code.

Notes are tagged with "note" icons that look like this.

Tips are tagged with "tip" icons that look like this.

When there is danger ahead, I warn you with a "caution" icon.

Preview features that are slated to become a part of the language or API in the future are labeled with this icon.

Java comes with a large programming library, or Application Programming Interface (API). When using an API call for the first time, I add a short summary description at the end of the section. These descriptions are a bit more informal but, hopefully, also a little more informative than those in the official online API documentation. The names of interfaces are in italics, just like in the official documentation. The number after a class, interface, or method name is the JDK version in which the feature was introduced, as shown in the following example:

java.lang.IO 25

- println(Object obj)
  Converts the object to a string and prints it on the console, followed by a line separator.

Programs whose source code is included in the companion code for this book are listed as examples, for instance

Listing **NotHelloWorld.java**

```
void main() {
   System.out.println("We will not use 'Hello, World!'");
}
```

You can download the companion code from https://horstmann.com/corejava.

Acknowledgment

Writing a book is always a monumental effort, and rewriting doesn't seem to be much easier, especially with such a rapid rate of change in Java technology. Making a book a reality takes many dedicated people, and it is my great pleasure to acknowledge the contributions of the entire Core Java team.

My thanks go to my editor, Harry Misthos, and to Julie Nahil from Pearson for steering the book through the production process. I wrote the book using HTML and CSS, and Prince (https://princexml.com) turned it into PDF—a workflow that I highly recommend.

Thanks to the many readers of earlier editions who reported embarrassing errors and made lots of thoughtful suggestions for improvement. I am particularly grateful to the excellent reviewing team that went over the manuscript with an amazing eye for detail and saved me from many more embarrassing errors.

Reviewers of this and earlier editions include Chuck Allison (Utah Valley University), Lance Andersen (Oracle), Gail Anderson (Anderson Software Group), Paul Anderson (Anderson Software Group), Alan Bateman (Oracle), Alec Beaton (IBM), Cliff Berg, Andrew Binstock (Oracle), Joshua Bloch, David Brown, Brian Burkhalter (Oracle), Corky Cartwright, Hillmer Chona, Frank Cohen (PushToTest), Chris Crane (devXsolution), Joe Darcy (Oracle), Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), Ahmad R. Elkomey, Hanno Embregts (Info Support), Robert Evans (Senior Staff, The Johns Hopkins University Applied Physics Lab), David Geary (Clarity Training), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon (Electric Cloud), Rob Gordon, John

Gray (University of Hartford), Cameron Gregory (olabs.com), Andrzej Grzesik, Marty Hall, Majid Hameed, Vincent Hardy (Adobe Systems), Dan Harkey (San Jose State University), Steve Haines, William Higgins (IBM), Marc Hoffmann (mtrail), Vladimir Ivanovic (PointBase), Jerry Jackson (CA Technologies), Josh Juneau, Heinz Kabutz (The Java Specialists' Newsletter, https://javaspecialists.eu), Stepan V. Kalinin (I-Teco/Servionica LTD), Tim Kimmet (Walmart), John Kostaras, Jerzy Krolak, Chris Laffra, Charlie Lai (Apple), Angelika Langer, Jeff Langr (Langr Software Solutions), Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Jason Lee (IBM), Gregory Longshore, Bob Lynch (Lynch Associates), Michael McMahon (Oracle), Rustam Mehmandarov, Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Maurice Naftalin, Mahesh Neelakanta (Florida Atlantic University), José Paumard (Oracle), Hao Pham, Paul Philion, Blake Ragsdell, Ylber Ramadani (Ryerson University), Stuart Reges (University of Arizona), Simon Ritter (Azul Systems), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and Brooks College), Naoto Sato (Oracle), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Yoshiki Shibata, Richard Slywczak (NASA/Glenn Research Center), Bradley A. Smith, Steven Stelting (Oracle), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (StreamingEdge), Janet Traub, Henri Tremblay, Paul Tyma (consultant), Christian Ullenboom, Peter van der Linden, Joe Wang (Oracle), Sven Woltmann, Burt Walsh, Dan Xu (Oracle), and John Zavgren (Oracle).

Finally, a warm thank you to my coauthor of earlier editions, Gary Cornell, and to Greg Doench who was my editor for almost thirty years.

Cay Horstmann
Düsseldorf, Germany
September 2025

Chapter 1 ▪ Streams

Compared to collections, streams provide a view of data that lets you specify computations at a higher conceptual level. With a stream, you specify what you want to have done, not how to do it. You leave the scheduling of operations to the implementation. For example, suppose you want to compute the average of a certain property. You specify the source of data and the property, and the stream library can then optimize the computation, for example by using multiple threads for computing sums and counts and combining the results.

In this chapter, you will learn how to use the Java stream library, which allows you to process sequences of values in a "what, not how" style.

1.1. From Iterating to Stream Operations

When you process a collection, you usually iterate over its elements and do some work with each of them. For example, suppose we want to count all long words in a book. First, let's put them into a list:

```
String contents =  Files.readString(Path.of("alice.txt")); // Read file int
String[] words = contents.split("\\PL+"); // Split into words
```

The split method splits a string into parts, given a regular expression for the delimiters. (See Chapter 2 for more information about regular expressions. For now, just take it on faith that the \PL+ expression works.)

Now we are ready to iterate:

```
int count = 0;
for (String w : words) {
   if (w.length() > 12) count++;
}
```

With streams, the same operation looks like this:

```
long count = Stream.of(words)
   .filter(w -> w.length() > 12)
   .count();
```

Now you don't have to scan the loop for evidence of filtering and counting. The method names tell you right away what the code intends to do. Moreover, where the loop prescribes the order of operations in complete detail, a stream is able to schedule the operations any way it wants, as long as the result is correct.

Simply changing stream to parallelStream allows the stream library to do the filtering and counting in parallel.

```
long count = words.parallelStream()
   .filter(w -> w.length() > 12)
   .count();
```

Streams follow the “what, not how” principle. In our stream example, we describe what needs to be done: get the long words and count them. We don't specify in which order, or in which thread, this should happen. In contrast, the loop at the beginning of this section specifies exactly how the computation should work, and thereby forgoes any chances of optimization.

A stream seems superficially similar to a collection, allowing you to transform and retrieve data. But there are significant differences:

1. A stream does not store its elements. They may be stored in an underlying collection or generated on demand.
2. Stream operations don't mutate their source. For

example, the filter method does not remove elements from a stream but yields a new stream in which they are not present.
3. Stream operations are lazy when possible. This means they are not executed until their result is needed. For example, if you only ask for the first five long words instead of all, the filter method will stop filtering after the fifth match. As a consequence, you can even have infinite streams!

Let us have another look at the example. The stream and parallelStream methods yield a stream for the words list. The filter method returns another stream that contains only the words of length greater than 12. The count method reduces that stream to a result.

This workflow is typical when you work with streams. You set up a pipeline of operations in three stages:

1. Create a stream.
2. Specify intermediate operations for transforming the initial stream into others, possibly in multiple steps.
3. Apply a terminal operation to produce a result. This operation forces the execution of the lazy operations that precede it. Afterwards, the stream can no longer be used.

In the example in Listing 1.1, the stream is created with the stream or parallelStream methods. The filter method transforms it, and count is the terminal operation.

In the next section, you will see how to create a stream. Three subsequent sections deal with intermediate operations. Then we turn to terminal operations.

Listing 1.1 **v2ch01/streams/CountLongWords.java**

```
package v2ch01.streams;

import module java.base;
```

```
class CountLongWords {
   void main() throws Exception {
      String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
      String[] words = contents.split("\\PL+");

      long count = 0;
      for (String w : words) {
         if (w.length() > 12) count++;
      }
      IO.println(count);

      count = Stream.of(words).filter(w -> w.length() > 12).count();
      IO.println(count);

      count = Stream.of(words).parallel().filter(w -> w.length() > 12).count();
      IO.println(count);
   }
}
```

java.util.stream.Stream<T>8

- Stream<T> filter(Predicate<? super T> p)
  yields a stream containing all elements of this stream fulfilling p.
- long count()
  yields the number of elements of this stream. This is a terminal operation.

java.util.Collection<E>1.2

- default Stream<E> stream()
- default Stream<E> parallelStream()
  yield a sequential or parallel stream of the elements in this collection.

1.2. Stream Creation

You have already seen that you can turn any collection into a stream with the stream method of the Collection interface. If

you have an array, use the static Stream.of method instead.

```
Stream<String> words = Stream.of(contents.split("\\PL+"));
```

In the preceding example, we make a stream from the array that the split method returns. The of method has a varargs parameter, so you can construct a stream from any number of arguments:

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Use Arrays.stream(array, from, to) to make a stream from a part of an array.

To make a stream with no elements, use the static Stream.empty method:

```
Stream<String> silence = Stream.empty();
   // Generic type <String> is inferred; same as Stream.<String>empty
```

The Stream interface has two static methods for making infinite streams. The generate method takes a function with no parameters (or, technically, an object of the Supplier<T> interface). Whenever a stream value is needed, that function is called to produce a value. You can get a stream of constant values as

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

or a stream of random numbers as

```
Stream<Double> randoms = Stream.generate(Math::random);
```

To produce sequences such as 0 1 2 3 . . ., use the iterate method instead. It takes a “seed” value and a function (technically, a UnaryOperator<T>) and repeatedly applies the function to the previous result. For example,

```
Stream<BigInteger> integers
   = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

The first element in the sequence is the seed BigInteger.ZERO.

The second element is f(seed) which yields 1 (as a big integer). The next element is f(f(seed)) which yields 2, and so on.

To produce a finite stream instead, add a predicate that specifies when the iteration should finish:

```
var limit = new BigInteger("10000000");
Stream<BigInteger> integers
   = Stream.iterate(BigInteger.ZERO,
      n -> n.compareTo(limit) < 0,
      n -> n.add(BigInteger.ONE));
```

As soon as the predicate rejects an iteratively generated value, the stream ends.

Finally, the Stream.ofNullable method makes a really short stream from an object. The stream has length 0 if the object is null or length 1 otherwise, containing just the object. This is mostly useful in conjunction with flatMap—see Section 1.7.7 for an example.

A number of methods in the Java API yield streams. For example, the String class has a lines method that yields a stream of the lines contained in the string:

```
Stream<String> greetings = "Hello\nGuten Tag\nBonjour".lines();
```

The Pattern class has a method splitAsStream that splits a CharSequence by a regular expression. You can use the following statement to split a string into words:

```
Stream<String> words = Pattern.compile("\\PL+").splitAsStream(cont
```

The Scanner.tokens method yields a stream of tokens of a scanner. Another way to get a stream of words from a string is

```
Stream<String> words = new Scanner(contents).tokens();
```

The static Files.lines method returns a Stream of all lines in a file:

```
try (Stream<String> lines = Files.lines(path)) {
```

```
    Process lines
}
```

Note that the try-with-resources block is necessary to close the file.

It is becoming more common to offer data as streams in the Java API. For example, Java 21 added a static availableLocales method that yields a stream of Locale objects.

To produce a finite stream, you can always add the elements to a list which you then turn into a stream. The Stream.Builder interface is slightly more efficient for that purpose:

```
Stream<Integer> digits(int n) {
   Stream.Builder<Integer> builder = Stream.builder();
   while (n != 0) {
      builder.add(n % 10);
      n = n / 10;
   }
   return builder.build();
}
```

To view the contents of one of the streams introduced in this section, use the toList method, which collects the stream's elements in a list. Like count, toList is a terminal operation. If the stream is infinite, first truncate it with the limit method:

```
IO.println(Stream.generate(Math::random).limit(10).toList());
```

In order to turn an Iterator, or a general Iterable that is not a collection, into a stream, you need to use helper methods for spliterators. A spliterator is similar to an iterator, but it can be split so that each resulting spliterator yields a part of the elements. This splitting is used for processing elements in parallel—see Section 1.13.

If you have an Iterator and want a stream of its results, use

```
StreamSupport.stream(Spliterators.spliteratorUnknownSize(
   iterator, Spliterator.ORDERED), false);
```

If you have an Iterable that is not a collection, you can turn it into a stream by calling

```
StreamSupport.stream(iterable.spliterator(), false);
```

It is very important that you don't modify the collection backing a stream while carrying out a stream operation. Remember that streams don't collect their data—the data is always in a separate collection. If you modify that collection, the outcome of the stream operations becomes undefined. The JDK documentation refers to this requirement as noninterference.

To be exact, since intermediate stream operations are lazy, it is possible to mutate the collection up to the point where the terminal operation executes. For example, the following, while certainly not recommended, will work:

```
List<String> wordList = . . .;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

But this code is wrong:

```
Stream<String> words = wordList.stream();
words.forEach(s -> {if (s.length() < 12) wordList.remove(s);});
   // ERROR--interference
```

The example program in Listing 1.2 shows the various ways of creating a stream.

Listing 1.2 **v2ch01/streams/CreatingStreams.java**

```
package v2ch01.streams;

import module java.base;

class CreatingStreams {
   void main() throws Exception {
      Path path = Path.of("gutenberg", "alice30.txt");
      String contents = Files.readString(path);
      Stream<String> words = Stream.of(contents.split("\\PL+"));

      show("words", words);
      Stream<String> song = Stream.of("gently", "down", "the", "stream");
      show("song", song);
      Stream<String> silence = Stream.empty();
      show("silence", silence);

      Stream<String> echos = Stream.generate(() -> "Echo");
      show("echos", echos);

      Stream<Double> randoms = Stream.generate(Math::random);
      show("randoms", randoms);

      Stream<BigInteger> integers = Stream.iterate(BigInteger.ONE,
            n -> n.add(BigInteger.ONE));
      show("integers", integers);

      Stream<String> greetings = "Hello\nGuten Tag\nBonjour".lines();
      show("greetings", greetings);

      Stream<String> wordsAnotherWay = Pattern.compile("\\PL+").splitAsStrea
      show("wordsAnotherWay", wordsAnotherWay);

      try (Stream<String> lines = Files.lines(path)) {
         show("lines", lines);
      }

      Iterable<Path> iterable = FileSystems.getDefault().getRootDirectories();
      Stream<Path> rootDirectories = StreamSupport.stream(iterable.spliterator
      show("rootDirectories", rootDirectories);

      Iterator<Path> iterator = Path.of("/usr/share/dict/words").iterator();
      Stream<Path> pathComponents = StreamSupport.stream(
            Spliterators.spliteratorUnknownSize(iterator, Spliterator.ORDERED), f
      show("pathComponents", pathComponents);
```

```
   }

   <T> void show(String title, Stream<T> stream) {
      final int SIZE = 10;
      List<T> firstElements = stream.limit(SIZE + 1).toList();
      IO.print(title + ": ");
      for (int i = 0; i < firstElements.size(); i++) {
         if (i > 0) IO.print(", ");
         if (i < SIZE)
            IO.print(firstElements.get(i));
         else
            IO.print("...");
      }
      IO.println();
   }

}
```

java.util.stream.Stream<T>8

- static <T> Stream<T> of(T... values)
  yields a stream whose elements are the given values.
- static <T> Stream<T> empty()
  yields a stream with no elements.
- static <T> Stream<T> generate(Supplier<T> s)
  yields an infinite stream whose elements are constructed by repeatedly invoking the function s.
- static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
- static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> f)
  yield a stream whose elements are seed, f invoked on seed, f invoked on the preceding element, and so on. The first method yields an infinite stream. The stream of the second method comes to an end before the first element that doesn't fulfill the hasNext predicate.
- static <T> Stream<T> ofNullable(T t)9
  returns an empty stream if t is null or a stream containing t otherwise.

java.util.Spliterators8

- static <T> Spliterator<T> spliteratorUnknownSize(Iterator<? extends T> iterator, int characteristics)
  turns an iterator into a splittable iterator of unknown size with the given characteristics (a bit pattern containing constants such as Spliterator.ORDERED).

java.util.Arrays 1.2

- static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive) 8
  yields a stream whose elements are the specified range of the array.

java.lang.String

- String[] split(String regex) 1.4
  finds all matches of the regular expression and splits the string by removing them. Returns an array with the fragments, but not a trailing empty string.
- static Stream<String> lines() 11
  yields a stream whose elements are the lines of this string.

java.util.regex.Pattern 1.4

- Stream<String> splitAsStream(CharSequence input) 8
  yields a stream whose elements are the parts of the input that are delimited by this pattern.

java.nio.file.Files 7

- static Stream<String> lines(Path path) 8
- static Stream<String> lines(Path path, Charset cs) 8
  yield a stream whose elements are the lines of the specified file, with the UTF-8 charset or the given charset.

java.util.stream.StreamSupport8

- static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel)
  yields a stream containing the values produced by the given splittable iterator.

java.lang.Iterable5.0

- Spliterator<T> spliterator()8
  yields a splittable iterator for this Iterable. The default implementation does not split and does not report a size.

java.util.Scanner5.0

- public Stream<String> tokens()9
  yields a stream of strings returned by calling the next method of this scanner.

java.util.function.Supplier<T>8

- T get()
  supplies a value.

1.3. The filter, map, and flatMap Methods

A stream transformation produces a stream whose elements are derived from those of another stream. You have already seen the filter transformation that yields a new stream with those elements that match a certain condition. Here, we transform a stream of strings into another stream containing only long words:

```
List<String> wordList = . . .;
Stream<String> longWords = wordList.stream().filter(w -> w.length()
```

The parameter type of filter is Predicate<T>—that is, a function from T to boolean.

Often, you want to transform the values in a stream in some way. Use the map method and pass the function that carries out the transformation. For example, you can transform all words to lowercase like this:

```
Stream<String> lowercaseWords = words.stream().map(String::toLow
```

Here, we used map with a method reference. Often, you will use a lambda expression instead:

```
Stream<Character> firstCodeUnits = words.stream().map(s -> s.charA
```

The resulting stream contains the first code unit of each word.

When you use map, a function is applied to each element, yielding a new stream of the returned values. Now consider the situation where the returned values are themselves streams. The following method yields a stream of all grapheme clusters of a string.

```
public static Stream<String> graphemeClusters(String s) {
   return Stream.of(s.split("\\b{g}"));
}
```

For example, graphemeClusters("Ahoy ☠") is a stream of strings "A", "h", "o", "y", " ", and "☠". (Note that the flag consists of multiple char values.)

Now let's map the graphemeClusters method on a stream of strings:

```
List<String> wordList = List.of(. . ., "your", "boat", . . .);
Stream<Stream<String>> result = wordList.stream().map(w -> graph
```

You will get a stream of streams, like [. . . ["y", "o", "u", "r"], ["b", "o", "a", "t"], . . .]. To flatten it out to a single stream [. . . "y", "o",

"u", "r", "b", "o", "a", "t", . . .], use the flatMap method instead of map:

```
Stream<String> flatResult = words.stream().flatMap(w -> graphemeC
   // Calls graphemeClusters on each word and flattens the results
```

Sometimes, it is inefficient to produce a stream for each result sequence. The mapMulti method offers an alternative. Instead of producing a stream of results, you generate the results and pass them to a collector—an object of a class implementing the functional interface Consumer. For each result, invoke the collector's accept method.

Let's do this with an example. The following loop iterates over the grapheme clusters of a string s:

```
BreakIterator iter = BreakIterator.getCharacterInstance();
. . .
iter.setText(s);
int start = iter.first();
int end = iter.next();
while (end != BreakIterator.DONE) {
   String gc = s.substring(start, end);
   start = end;
   end = iter.next();
   // Do something with gc
}
```

When calling mapMulti, you provide a function that is invoked with the stream element and the collector. In your function, pass your results to the collector.

```
Stream<String> result = words.stream().mapMulti((s, collector) -> {
   iter.setText(s);
   int start = iter.first();
   int end = iter.next();
   while (end != BreakIterator.DONE) {
      String gc = s.substring(start, end);
      start = end;
      end = iter.next();
      collector.accept(gc);
```

```
    }
});
```

java.util.stream.Stream<T>[8]

- Stream<T> filter(Predicate<? super T> predicate)
  yields a stream containing the elements of this stream that fulfill the predicate.
- <R> Stream<R> map(Function<? super T,? extends R> mapper)
  yields a stream containing the results of applying mapper to the elements of this stream.
- <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
  yields a stream obtained by concatenating the results of applying mapper to the elements of this stream. (Note that each mapper result is a stream.)
- <R> Stream<R> mapMulti(BiConsumer<? super T,? super Consumer<R>> mapper)[16]
  For each stream element, the mapper is invoked, and all elements passed to the Consumer during invocation are added to the result stream.

1.4. Extracting Substreams and Combining Streams

The call stream.limit(n) returns a new stream that ends after n elements (or when the original stream ends if it is shorter). This method is particularly useful for cutting infinite streams down to size. For example,

```
Stream<Double> randoms = Stream.generate(Math::random).limit(1
```

yields a stream with 100 random numbers.

The call stream.skip(n) does the exact opposite. It discards the first n elements. This is handy in our book reading example

where, due to the way the split method works, the first element is an unwanted empty string. We can make it go away by calling skip:

```
Stream<String> words = Stream.of(contents.split("\\PL+")).skip(1);
```

The *stream*.takeWhile(*predicate*) call takes all elements from the stream while the predicate is true, and then stops.

For example, suppose we use the graphemeClusters method of the preceding section to split a string into characters, and we want to collect all initial digits. The takeWhile method can do this:

```
Stream<String> initialDigits = graphemeClusters(str).takeWhile(
   s -> "0123456789".contains(s));
```

The dropWhile method does the opposite, dropping elements while a condition is true and yielding a stream of all elements starting with the first one for which the condition was false. For example,

```
Stream<String> withoutInitialWhiteSpace = graphemeClusters(str).d
   s -> s.strip().length() == 0);
```

You can concatenate two streams with the static concat method of the Stream class:

```
Stream<String> combined = Stream.concat(
   graphemeClusters("Hello"), graphemeClusters("World"));
   // Yields the stream ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

Of course, the first stream should not be infinite—otherwise the second one wouldn't ever get a chance.

java.util.stream.Stream<T>8

- Stream<T> limit(long maxSize)
  yields a stream with up to maxSize of the initial elements from this stream.
- Stream<T> skip(long n)

yields a stream whose elements are all but the initial n elements of this stream.

- Stream<T> takeWhile(Predicate<? super T> predicate)9
  yields a stream whose elements are the initial elements of this stream that fulfill the predicate.
- Stream<T> dropWhile(Predicate<? super T> predicate)9
  yields a stream whose elements are the elements of this stream except for the initial ones that fulfill the predicate.
- static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
  yields a stream whose elements are the elements of a followed by the elements of b.

1.5. Other Stream Transformations

The distinct method returns a stream that yields elements from the original stream, in the same order, except that duplicates are suppressed. The duplicates need not be adjacent.

```
Stream<String> uniqueWords
   = Stream.of("merrily", "merrily", "merrily", "gently").distinct();
   // Only one "merrily" is retained
```

For sorting a stream, there are several variations of the sorted method. One works for streams of Comparable elements, and another accepts a Comparator. Here, we sort strings so that the longest ones come first:

```
Stream<String> longestFirst
   = words.stream().sorted(Comparator.comparing(String::length).rev
```

As with all stream transformations, the sorted method yields a new stream whose elements are the elements of the original stream in sorted order.

Of course, you can sort a collection without using streams. The sorted method is useful when the sorting process is part of a stream pipeline.

Finally, the peek method yields another stream with the same elements as the original, but a function is invoked every time an element is retrieved. That is handy for debugging:

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
   .peek(e -> IO.println("Fetching " + e))
   .limit(20)
   .toArray();
```

When an element is actually accessed, a message is printed. This way you can verify that the infinite stream returned by iterate is processed lazily.

When you use a debugger to debug a stream computation, you can set a breakpoint in a method that is called from one of the transformations. With most IDEs, you can also set breakpoints in lambda expressions. If you just want to know what happens at a particular point in the stream pipeline, add

```
.peek(x -> {
   return;
})
```

and set a breakpoint on the second line.

---

java.util.stream.Stream<T>[8]

- Stream<T> distinct()
  yields a stream of the distinct elements of this stream.
- Stream<T> sorted()
- Stream<T> sorted(Comparator<? super T> comparator)
  yield a stream whose elements are the elements of this stream in sorted order. The first method requires that the elements are instances of a class implementing Comparable.
- Stream<T> peek(Consumer<? super T> action)
  yields a stream with the same elements as this stream, passing each element to action as it is consumed.

1.6. Simple Reductions

Now that you have seen how to create and transform streams, we will finally get to the most important point—getting answers from the stream data. The methods covered in this section are called reductions. Reductions are terminal operations. They reduce the stream to a nonstream value that can be used in your program.

You have already seen a simple reduction: the count method that returns the number of elements of a stream.

Other simple reductions are max and min that return the largest or smallest value. There is a twist—these methods return an Optional<T> value that either wraps the answer or indicates that there is none (because the stream happened to be empty). In the olden days, it was common to return null in such a situation. But that can lead to null pointer exceptions when it happens in an incompletely tested program. The Optional type is a better way of indicating a missing return value. We discuss the Optional type in detail in the next section. Here is how you can get the maximum of a stream:

```
Optional<String> largest = words.max(String::compareToIgnoreCase
IO.println("largest: " + largest.orElse(""));
```

The findFirst returns the first value in a nonempty collection. It is often useful when combined with filter. For example, here we find the first word that starts with the letter Q, if it exists:

```
Optional<String> startsWithQ
   = words.filter(s -> s.startsWith("Q")).findFirst();
```

If you are OK with any match, not just the first one, use the findAny method. This is effective when you parallelize the stream, since the stream can report any match that it finds instead of being constrained to the first one.

```
Optional<String> startsWithQ
   = words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

If you just want to know if there is a match, use the terminal anyMatch operation with a predicate argument:

```
boolean aWordStartsWithQ
   = words.parallel().anyMatch(s -> s.startsWith("Q"));
```

There are methods allMatch and noneMatch that return true if all or no elements match a predicate. These methods also benefit from being run in parallel.

java.util.stream.Stream<T>8

- Optional<T> max(Comparator<? super T> comparator)
- Optional<T> min(Comparator<? super T> comparator)
  yield a maximum or minimum element of this stream, using the ordering defined by the given comparator, or an empty Optional if this stream is empty. These are terminal operations.
- Optional<T> findFirst()
- Optional<T> findAny()
  yield the first, or any, element of this stream, or an empty Optional if this stream is empty. These are terminal operations.
- boolean anyMatch(Predicate<? super T> predicate)
- boolean allMatch(Predicate<? super T> predicate)
- boolean noneMatch(Predicate<? super T> predicate)
  return true if any, all, or none of the elements of this stream match the given predicate. These are terminal operations.

1.7. The Optional Type

An Optional<T> object is a wrapper for either an object of type T or no object. In the former case, we say that the value is present. The Optional<T> type is intended as a safer alternative for a reference of type T that either refers to an object or is null. But it is only safer if you use it right. The next three sections show you how.

1.7.1. Getting an Optional Value

The key to using Optional effectively is to use a method that either produces an alternative if the value is not present, or consumes the value only if it is present.

In this section, we look at the first strategy. Often, there is a default that you want to use when there is no match, perhaps the empty string:

```
String result = optionalString.orElse("");
   // The wrapped string, or "" if none
```

You can also invoke code to compute the default:

```
String result = optionalString.orElseGet(() -> System.getProperty("my
   // The function is only called when needed
```

Or you can throw an exception if there is no value:

```
String result = optionalString.orElseThrow(IllegalStateException::new
   // Supply a method that yields an exception object
```

java.util.Optional8

- T orElse(T other)
  yields the value of this Optional, or other if this Optional is empty.
- T orElseGet(Supplier<? extends T> other)
  yields the value of this Optional, or the result of invoking other if this Optional is empty.
- <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
  yields the value of this Optional, or throws the result of invoking exceptionSupplier if this Optional is empty.

1.7.2. Consuming an Optional Value

In the preceding section, you saw how to produce an alternative if no value is present. The other strategy for working with optional values is to consume the value only if it is present.

The ifPresent method accepts a function. If the optional value exists, it is passed to that function. Otherwise, nothing happens.

```
optionalValue.ifPresent(v -> Process v);
```

For example, if you want to add the value to a set if it is present, call

```
optionalValue.ifPresent(v -> results.add(v));
```

or simply

```
optionalValue.ifPresent(results::add);
```

If you want to take one action if the Optional has a value and another action if it doesn't, use ifPresentOrElse:

```
optionalValue.ifPresentOrElse(
   v -> IO.println("Found " + v),
   () -> logger.warning("No match"));
```

java.util.Optional[8]

- void ifPresent(Consumer<? super T> action)
  If this Optional is nonempty, passes its value to action.
- void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)[9]
  If this Optional is nonempty, passes its value to action, else invokes emptyAction.

1.7.3. Pipelining Optional Values

In the preceding sections, you saw how to get a value out of an Optional object. Another useful strategy is to keep the Optional

intact. You can transform the value inside an Optional by using the map method:

```
Optional<Path> transformed = optionalString.map(Path::of);
```

If optionalString is empty, then transformed is also empty.

> This map method is the analog of the map method of the Stream interface that you have seen in Section 1.3. Simply imagine an optional value as a stream of size zero or one. The result again has size zero or one, and in the latter case, the function has been applied.

Similarly, you can use the filter method to only consider Optional values that fulfill a certain property before or after transforming it. If the property is not fulfilled, the pipeline yields an empty result:

```
Optional<Path> transformed = optionalString
   .filter(s -> s.endsWith(".txt"))
   .map(Path::of);
```

You can substitute an alternative Optional for an empty Optional with the or method. The alternative is computed lazily.

```
Optional<String> result = optionalString.or(() -> // Supply an Optiona
   alternatives.stream().findFirst());
```

If optionalString has a value, then result is optionalString. If not, the lambda expression is evaluated, and its result is used.

java.util.Optional8
■ <U> Optional<U> map(Function<? super T,? extends U> mapper) yields an Optional whose value is obtained by applying the given function to the value of this Optional if present, or an empty Optional otherwise.

- Optional<T> filter(Predicate<? super T> predicate)
  yields an Optional with the value of this Optional if it fulfills the given predicate, or an empty Optional otherwise.
- Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)[9]
  yields this Optional if it is nonempty, or the one produced by the supplier otherwise.

1.7.4. How Not to Work with Optional Values

If you don't use Optional values correctly, you have no benefit over the “something or null” approach of the past.

The get method gets the wrapped element of an Optional value if it exists, or throws a NoSuchElementException if it doesn’t. Therefore,

```
Optional<T> optionalValue = . . .;
optionalValue.get().someMethod()
```

is no safer than

```
T value = . . .;
value.someMethod();
```

The isPresent and isEmpty methods report whether or not an Optional<T> object has a value. But

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

is no easier than

```
if (value != null) value.someMethod();
```

If you find yourself using the get, isPresent, or isEmpty methods, try to rethink your approach and use one of the strategies of the preceding two sections.

The orElseThrow method is a scarier-sounding synonym for the get method. When you call optionalValue.orElseThrow().someMethod(), it becomes explicit that an exception will occur if the optionalValue is empty. The hope is that programmers will only use orElseThrow when it is absolutely clear that this cannot happen.

Here are a few more tips for the proper use of the Optional type:

- A variable of type Optional should never be null.
- Don't use fields of type Optional. The cost is an additional object. Inside a class, use null for an absent field. To discourage Optional fields, the class is not serializable.
- Method parameters of type Optional are questionable. They make the call unpleasant in the common case where the requested value is present. Instead, consider two overloaded versions of the method, with and without the parameter. (On the other hand, returning an Optional is fine. It is the proper way to indicate that a function may not have a result.)
- Don't put Optional objects in a set, and don't use them as keys for a map. Collect the values instead.

java.util.Optional[8]

- T get()
- T orElseThrow()[10]
  yield the value of this Optional, or throw a NoSuchElementException if it is empty.
- boolean isEmpty()[11]
- boolean isPresent()
  return true if this Optional is empty or not empty.

1.7.5. Creating Optional Values

So far, we have discussed how to consume an Optional object

someone else created. If you want to write a method that creates an Optional object, there are several static methods for that purpose, including Optional.of(result) and Optional.empty(). For example,

```
public static Optional<Double> inverse(Double x) {
   return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

The ofNullable method is intended as a bridge from possibly null values to optional values. Optional.ofNullable(obj) returns Optional.of(obj) if obj is not null and Optional.empty() otherwise.

java.util.Optional8

- static <T> Optional<T> of(T value)
- static <T> Optional<T> ofNullable(T value)
  yield an Optional with the given value. If value is null, the first method throws a NullPointerException and the second method yields an empty Optional.
- static <T> Optional<T> empty()
  yields an empty Optional.

1.7.6. Composing Optional Value Functions with flatMap

Suppose you have a method f yielding an Optional<T>, and the target type T has a method g yielding an Optional<U>. If they were normal methods, you could compose them by calling s.f().g(). But that composition doesn't work since s.f() has type Optional<T>, not T. Instead, call

```
Optional<U> result = s.f().flatMap(T::g);
```

If s.f() is present, then g is applied to it. Otherwise, an empty Optional<U> is returned.

Clearly, you can repeat that process if you have more methods or lambdas that yield Optional values. You can then build a pipeline of steps, simply by chaining calls to flatMap, that will succeed only when all parts do.

For example, consider the safe inverse method of the preceding section. Suppose we also have a safe square root:

```
public static Optional<Double> squareRoot(Double x) {
   return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

Then you can compute the square root of the inverse as

```
Optional<Double> result = inverse(arg).flatMap(x -> squareRoot(x));
```

We can make such a chain look a little more regular by starting out with an Optional:

```
Optional<Double> result
   = Optional.of(arg).flatMap(x -> inverse(x)).flatMap(x -> squareRoot
```

It looks even neater with method expressions:

```
Optional<Double> result
   = Optional.of(arg).flatMap(this::inverse).flatMap(this::squareRoot)
```

If either the inverse method or the squareRoot returns Optional.empty(), the result is empty.

In this example, taken from Listing 1.3, the inverse and squareRoot methods are instance methods. Hence the method expressions start with this:: . Had these been static methods, they would be prefixed by *ClassName*::.

You have already seen a flatMap method in the Stream interface (see Section 1.3). That method maps a stream-yielding method to all elements of a stream, and then flattens out the resulting stream of streams. The Optional.flatMap method works in the same way. By

---

mapping an optional-yielding method to an optional, you get an optional of optional, which is then flattened out.

java.util.Optional8

- <U> Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper)
  yields the result of applying mapper to the value in this Optional if present, or an empty optional otherwise.

1.7.7. Turning an Optional into a Stream

The stream method turns an Optional<T> into a Stream<T> with zero or one element. Sure, why not, but why would you ever want that?

This becomes useful with methods that return an Optional result. Suppose you have a stream of user IDs and a method

```
Optional<User> lookup(String id)
```

How do you get a stream of users, skipping those IDs that are invalid?

Of course, you can filter out the invalid IDs and then apply get to the remaining ones:

```
Stream<String> ids = . . .;
Stream<User> users = ids.map(Users::lookup)
   .filter(Optional::isPresent)
   .map(Optional::get);
```

But that uses the isPresent and get methods that we warned about. It is more elegant to call

```
Stream<User> users = ids.map(Users::lookup)
   .flatMap(Optional::stream);
```

Each call to stream returns a stream with zero or one element.

The flatMap method combines them all. That means the nonexistent users are simply dropped.

In this section, we consider the happy scenario in which we have a method that returns an Optional value. These days, many methods return null when there is no valid result. Suppose Users.classicLookup(id) returns a User object or null, not an Optional<User>. Then you can of course filter out the null values:

```
Stream<User> users = ids.map(Users::classicLookup)
   .filter(Objects::nonNull);
```

But if you prefer the flatMap approach, you can use

```
Stream<User> users = ids.flatMap(
   id -> Stream.ofNullable(Users.classicLookup(id)));
```

or

```
Stream<User> users = ids.map(Users::classicLookup)
   .flatMap(Stream::ofNullable);
```

The call Stream.ofNullable(obj) yields an empty stream if obj is null or a stream just containing obj otherwise.

The example program in Listing 1.3 demonstrates the Optional API.

Listing 1.3 **v2ch01/optional/OptionalDemo.java**

```
package v2ch01.optional;

import module java.base;

class OptionalDemo {
   void main() throws Exception {
      String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
      List<String> wordList = List.of(contents.split("\\PL+"));
```

```
    Optional<String> optionalValue = wordList
            .stream()
            .filter(s -> s.contains("fred"))
            .findFirst();
    IO.println(optionalValue.orElse("No word") + " contains fred");

    Optional<String> optionalString = Optional.empty();
    String result = optionalString.orElse("N/A");
    IO.println("result: " + result);
    result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName(
    IO.println("result: " + result);
    try {
        result = optionalString.orElseThrow(IllegalStateException::new);
        IO.println("result: " + result);
    }
    catch (Throwable t) {
        t.printStackTrace(System.out);
    }

    optionalValue = wordList.stream().filter(s -> s.contains("red")).findFirst();
    optionalValue.ifPresent(s -> IO.println(s + " contains red"));

    var results = new HashSet<String>();
    optionalValue.ifPresent(results::add);
    Optional<Boolean> added = optionalValue.map(results::add);
    IO.println(added);

    IO.println(inverse(4.0).flatMap(this::squareRoot));
    IO.println(inverse(-1.0).flatMap(this::squareRoot));
    IO.println(inverse(0.0).flatMap(this::squareRoot));
    Optional<Double> result2 = Optional.of(-4.0)
            .flatMap(this::inverse)
            .flatMap(this::squareRoot);
    IO.println(result2);
}

Optional<Double> inverse(Double x) {
    return x == 0 ? Optional.empty() : Optional.of(1 / x);
}

Optional<Double> squareRoot(Double x) {
    return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

}

java.util.Optional[8]

- <U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)[9]
  yields the result of applying mapper to the value of this Optional, or an empty Optional if this Optional is empty.

1.8. Collecting Results

When you are done with a stream, you will often want to look at the results. You can call the iterator method, which yields an old-fashioned iterator that you can use to visit the elements.

Alternatively, you can call the forEach method to apply a function to each element:

```
stream.forEach(IO::println);
```

On a parallel stream, the forEach method traverses elements in arbitrary order. If you want to process them in stream order, call forEachOrdered instead. Of course, you might then give up some or all of the benefits of parallelism.

But more often than not, you will want to collect the result in a data structure. You have already seen the toList method that yields a list of the stream elements.

Call toArray to get an array of the stream elements.

Since it is not possible to create a generic array at runtime, the expression stream.toArray() returns an Object[] array. If you want an array of the correct type, pass in the array constructor:

```
String[] result = stream.toArray(String[]::new);
   // stream.toArray() has type Object[]
```

To collect results into other data structures, the Stream interface has a generic collect method. You can concatenate the

results in a string, store them in any collection, and insert them into maps. By implementing your own collector, you can consume the stream data in arbitrary ways. You will see the details in the following sections.

java.util.stream.BaseStream8

- Iterator<T> iterator()
  yields an iterator for obtaining the elements of this stream. This is a terminal operation.

java.util.stream.Stream<T>8

- List<T> toList()16
  yields a list of the elements in this stream.
- void forEach(Consumer<? super T> action)
  invokes action on each element of the stream.
- Object[] toArray()
- <A> A[] toArray(IntFunction<A[]> generator)
  yield an array of objects, or of type A when passed a constructor reference A[]::new. These are terminal operations.
- <R,A> R collect(Collector<? super T,A,R> collector)
  collects the elements in this stream, using the given collector. The Collectors class has factory methods for many collectors.

1.9. Collectors

For collecting stream elements to another target, there is a convenient collect method that takes an instance of the Collector interface. A collector is an object that accumulates elements and produces a result. The Collectors class provides a large number of factory methods for common collectors.

Suppose you want to collect all strings in a stream by

concatenating them. You can call

```
String result = stream.collect(Collectors.joining());
```

If you want a delimiter between elements, pass it to the joining method:

```
String result = stream.collect(Collectors.joining(", "));
```

If your stream contains objects other than strings, you need to first convert them to strings, like this:

```
String result = stream.map(Object::toString).collect(Collectors.joining
```

If you want to reduce the stream results to a sum, count, average, maximum, or minimum, use one of the summarizing(Int|Long|Double) methods. These methods take a function that maps the stream objects to numbers and yield a result of type (Int|Long|Double)SummaryStatistics, simultaneously computing the sum, count, average, maximum, and minimum.

```
IntSummaryStatistics summary = stream.collect(
   Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

Note that you are not calling the joining and summarizingInt methods on a stream. These are static methods of the Collectors class. They yield instances of Collector which you pass to the collect method of the Stream interface.

The joining collector yields a string, and the summarizing collectors yield four numbers. However, most collectors collect values in data structures, as you will see in the following sections.

The example program in Listing 1.5 shows how to collect elements from a stream.

Listing 1.4 **v2ch01/collecting/CollectingResults.java**

```
package v2ch01.collecting;

import module java.base;

class CollectingResults {
   void main() throws Exception {
      // Turning a stream into an iterator
      Iterator<Integer> iter = Stream.iterate(0, n -> n + 1).limit(10).iterator();
      while (iter.hasNext())
         IO.println(iter.next());

      Object[] numbers = Stream.iterate(0, n -> n + 1).limit(10).toArray();
      IO.println("Object array: " + numbers + " " + Arrays.toString(numbers));
      // Note it's an Object[] array

      try {
         var number = (Integer) numbers[0]; // OK
         IO.println("number: " + number);
         IO.println("The following statement throws an exception:");
         var _ = (Integer[]) numbers; // Throws exception
      }
      catch (ClassCastException e) {
         IO.println(e);
      }

      Integer[] numbers3 = Stream.iterate(0, n -> n + 1).limit(10).toArray(Integer
      IO.println("Integer array: " + numbers3 + " " + Arrays.toString(numbers3))
      // Note it's an Integer[] array

      String result = noVowels().limit(10).collect(Collectors.joining());
      IO.println("Joining: " + result);
      result = noVowels().limit(10).collect(Collectors.joining(", "));
      IO.println("Joining with commas: " + result);

      IntSummaryStatistics summary = noVowels()
            .collect(Collectors.summarizingInt(String::length));
      double averageWordLength = summary.getAverage();
      double maxWordLength = summary.getMax();
      IO.println("Average word length: " + averageWordLength);
      IO.println("Max word length: " + maxWordLength);
      IO.println("forEach:");
      noVowels().limit(10).forEach(IO::println);
```

```
   }

   Stream<String> noVowels() throws IOException {
      String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
      Stream<String> words = Stream.of(contents.split("\\PL+"));
      return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
   }

   <T> void show(String label, Set<T> set) {
      IO.print(label + ": " + set.getClass().getName());
      IO.println("["
         + set.stream().limit(10).map(Object::toString).collect(Collectors.joining(",
         + "]");
   }
}
```

java.util.stream.Collectors8

- static Collector<CharSequence,?,String> joining()
- static Collector<CharSequence,?,String> joining(CharSequence delimiter)
- static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)
  yield a collector that joins strings. The delimiter is placed between strings, and the prefix and suffix before the first and after the last string. When not specified, these are empty.
- static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)
- static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)
- static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)
  yield collectors that produce an (Int|Long|Double)SummaryStatistics object, from which you can obtain the count, sum, average, maximum, and minimum of the results of applying mapper to each element.

java.util.IntSummaryStatistics8

java.util.LongSummaryStatistics8

java.util.DoubleSummaryStatistics8

- long getCount()
  yields the count of the summarized elements.
- *xxx* getSum()
- double getAverage()
  yield the sum or average of the summarized elements, or zero if there are no elements.
- *xxx* getMax()
- *xxx* getMin()
  yield the maximum or minimum of the summarized elements, or (Integer|Long|Double).(MAX|MIN)_VALUE if there are no elements.

1.9.1. Collecting into Collections

Before the toList method was added in Java 16, you had to use the collector produced by Collectors.toList():

```
List<String> result = stream.collect(Collectors.toList());
```

Similarly, here is how you can collect stream elements into a set:

```
Set<String> result = stream.collect(Collectors.toSet());
```

These calls give you a list or set, but you cannot make any further assumptions. The collection might not be mutable, serializable, or thread-safe. If you want to control which kind of collection you get, use the following call instead:

```
TreeSet<String> result = stream.collect(Collectors.toCollection(TreeS
```

The example program in Listing 1.5 shows how to place stream

elements into a collection.

Listing 1.5 **v2ch01/collecting/CollectingResults.java**

```
package v2ch01.collecting;

import module java.base;

class CollectingResults {
   void main() throws Exception {
      // Turning a stream into an iterator
      Iterator<Integer> iter = Stream.iterate(0, n -> n + 1).limit(10).iterator();
      while (iter.hasNext())
         IO.println(iter.next());

      Object[] numbers = Stream.iterate(0, n -> n + 1).limit(10).toArray();
      IO.println("Object array: " + numbers + " " + Arrays.toString(numbers));
      // Note it's an Object[] array

      try {
         var number = (Integer) numbers[0]; // OK
         IO.println("number: " + number);
         IO.println("The following statement throws an exception:");
         var _ = (Integer[]) numbers; // Throws exception
      }
      catch (ClassCastException e) {
         IO.println(e);
      }

      Integer[] numbers3 = Stream.iterate(0, n -> n + 1).limit(10).toArray(Integer
      IO.println("Integer array: " + numbers3 + " " + Arrays.toString(numbers3))
      // Note it's an Integer[] array

      String result = noVowels().limit(10).collect(Collectors.joining());
      IO.println("Joining: " + result);
      result = noVowels().limit(10).collect(Collectors.joining(", "));
      IO.println("Joining with commas: " + result);

      IntSummaryStatistics summary = noVowels()
            .collect(Collectors.summarizingInt(String::length));
      double averageWordLength = summary.getAverage();
      double maxWordLength = summary.getMax();
```

```
      IO.println("Average word length: " + averageWordLength);
      IO.println("Max word length: " + maxWordLength);
      IO.println("forEach:");
      noVowels().limit(10).forEach(IO::println);
   }

   Stream<String> noVowels() throws IOException {
      String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
      Stream<String> words = Stream.of(contents.split("\\PL+"));
      return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
   }

   <T> void show(String label, Set<T> set) {
      IO.print(label + ": " + set.getClass().getName());
      IO.println("["
         + set.stream().limit(10).map(Object::toString).collect(Collectors.joining(",
         + "]");
   }
}
```

java.util.stream.Collectors[8]

- static <T> Collector<T,?,List<T>> toList()
- static <T> Collector<T,?,List<T>> toUnmodifiableList()[10]
- static <T> Collector<T,?,Set<T>> toSet()
- static <T> Collector<T,?,Set<T>> toUnmodifiableSet()[10]
  yield collectors that collect elements in a list or set.
- static <T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory)
  yields a collector that collects elements into an arbitrary collection. Pass a constructor reference such as TreeSet::new.

1.9.2. Collecting into Maps

Suppose you have a Stream<Person> and want to collect the elements into a map so that later you can look up people by their ID. Call Collectors.toMap with two functions that produce the map's keys and values. For example,

```
public record Person(int id, String name) {}
. . .
Map<Integer, String> idToName = people.collect(
   Collectors.toMap(Person::id, Person::name));
```

In the common case when the values should be the actual elements, use Function.identity() for the second function.

```
Map<Integer, Person> idToPerson = people.collect(
   Collectors.toMap(Person::id, Function.identity()));
```

If there is more than one element with the same key, there is a conflict, and the collector will throw an IllegalStateException. You can override that behavior by supplying a third function that resolves the conflict and determines the value for the key, given the existing and the new value. Your function could return the existing value, the new value, or a combination of them.

Here, we construct a map that contains, for each language in the available locales, as key its name in your default locale (such as "German"), and as value its localized name (such as "Deutsch").

```
Map<String, String> languageNames = Locale.availableLocales().coll
   Collectors.toMap(
      Locale::getDisplayLanguage,
      loc -> loc.getDisplayLanguage(loc),
      (existingValue, newValue) -> existingValue));
```

We don't care that the same language might occur twice (for example, German in Germany and in Switzerland), so we just keep the first entry.

> In this chapter, I use the Locale class as a source of an interesting data set. See Chapter 7 for more information on working with locales.

Now suppose we want to know all languages in a given country. Then we need a Map<String, Set<String>>. For example, the

value for "Switzerland" is the set [French, German, Italian]. At first, we store a singleton set for each language. Whenever a new language is found for a given country, we form the union of the existing and the new set.

```
Map<String, Set<String>> countryLanguageSets = Locale.availableLo
   Collectors.toMap(
      Locale::getDisplayCountry,
      l -> Collections.singleton(l.getDisplayLanguage()),
      (a, b) -> { // Union of a and b
         var union = new HashSet<String>(a);
         union.addAll(b);
         return union;
      }));
```

You will see a simpler way of obtaining this map in the next section.

If you want a TreeMap, supply the constructor as the fourth argument. You must provide a merge function. Here is one of the examples from the beginning of the section, now yielding a TreeMap:

```
Map<Integer, Person> idToPerson = people.collect(
   Collectors.toMap(
      Person::id,
      Function.identity(),
      (existingValue, newValue) -> { throw new IllegalStateException(
      TreeMap::new));
```

For each of the toMap methods, there is an equivalent toConcurrentMap method that yields a concurrent map. A single concurrent map is used in the parallel collection process. When used with a parallel stream, a shared map is more efficient than merging maps. Note that elements are no longer collected in stream order, but that doesn't usually make a difference.

The program in Listing 1.6 gives examples of collecting stream results into maps.

Listing 1.6 **v2ch01/collecting/CollectingIntoMaps.java**

```
package v2ch01.collecting;

import module java.base;

record Person(int id, String name) {}

class CollectingIntoMaps {
   void main() throws Exception {
      Map<Integer, String> idToName = people()
            .collect(Collectors.toMap(Person::id, Person::name));
      IO.println("idToName: " + idToName);

      Map<Integer, Person> idToPerson = people()
            .collect(Collectors.toMap(Person::id, Function.identity()));
      IO.println("idToPerson: " + idToPerson.getClass().getName() + idToPerson);

      idToPerson = people().collect(
            Collectors.toMap(Person::id, Function.identity(), (_, _) -> {
               throw new IllegalStateException();
            }, TreeMap::new));
      IO.println("idToPerson: " + idToPerson.getClass().getName() + idToPerson);

      Map<String, String> languageNames = Locale.availableLocales()
            .collect(Collectors.toMap(Locale::getDisplayLanguage,
                  l -> l.getDisplayLanguage(l),
                  (existingValue, _) -> existingValue));
      IO.println("languageNames: " + languageNames);

      Map<String, Set<String>> countryLanguageSets = Locale.availableLocales(
            .collect(Collectors.toMap(Locale::getDisplayCountry,
                  l -> Set.of(l.getDisplayLanguage()), (a, b) -> { // union of a and b
                     Set<String> union = new HashSet<>(a);
                     union.addAll(b);
                     return union;
                  }));
      IO.println("countryLanguageSets: " + countryLanguageSets);
   }

   static Stream<Person> people() {
      return Stream.of(new Person(1001, "Peter"), new Person(1002, "Paul"),
```

```
            new Person(1003, "Mary"));
    }
}
```

java.util.stream.Collectors8

- static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)
- static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)
- static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)
- static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)10
- static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)10
- static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)
- static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)
- static <T,K,U,M extends ConcurrentMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)
  yield a collector that produces a map, unmodifiable map, or concurrent map. The keyMapper and valueMapper

functions are applied to each collected element, yielding a key/value entry of the resulting map. By default, an IllegalStateException is thrown when two elements give rise to the same key. You can instead supply a mergeFunction that merges values with the same key. By default, the result is a HashMap or ConcurrentHashMap. You can instead supply a mapSupplier that yields the desired map instance.

1.9.3. Grouping and Partitioning

In the preceding section, you saw how to collect all languages in a given country. But the process was a bit tedious. You had to generate a singleton set for each map value and then specify how to merge the existing and new values. Forming groups of values with the same characteristic is very common, so the groupingBy method supports it directly.

Let's look at the problem of grouping locales by country. First, form this map:

```
Map<String, List<Locale>> countryToLocales = Locale.availableLoca
   Collectors.groupingBy(Locale::getCountry));
```

The function Locale::getCountry is the classifier function of the grouping. You can now look up all locales for a given country code, for example

```
List<Locale> swissLocales = countryToLocales.get("CH");
   // Yields locales de_CH, fr_CH, it_CH, and maybe more
```

A quick refresher on locales: Each locale has a language code (such as en for English) and a country code (such as US for the United States). The locale en_US describes English in the United States, and en_IE is English in Ireland. Some countries have multiple locales. For example, ga_IE is Gaelic in Ireland, and, as the preceding example shows, the JDK knows at least three locales in

---

Switzerland.

When the classifier function is a predicate function (that is, a function returning a boolean value), the stream elements are partitioned into two lists: those where the function returns true and the complement. In this case, it is more efficient to use partitioningBy instead of groupingBy. For example, here we split all locales into those that use English and all others:

```
Map<Boolean, List<Locale>> englishAndOtherLocales = Locale.avail
   Collectors.partitioningBy(l -> l.getLanguage().equals("en")));
List<Locale> englishLocales = englishAndOtherLocales.get(true);
```

If you call the groupingByConcurrent method, you get a concurrent map that, when used with a parallel stream, is concurrently populated. This is entirely analogous to the toConcurrentMap method.

java.util.stream.Collectors8

- static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)
- static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)
  yield a collector that produces a map or concurrent map whose keys are the results of applying classifier to all collected elements, and whose values are lists of elements with the same key.
- static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)
  yields a collector that produces a map whose keys are true/false, and whose values are lists of the elements that fulfill/do not fulfill the predicate.

1.9.4. Downstream Collectors

The groupingBy method yields a map whose values are lists. If you want to process those lists in some way, supply a downstream collector. For example, if you want sets instead of lists, you can use the Collectors.toSet collector that you saw in the preceding section:

```
Map<String, Set<Locale>> countryToLocaleSet = Locale.availableLoc
   groupingBy(Locale::getCountry, toSet()));
```

> In this example, as well as the remaining examples of this section, I assume a static import of java.util.stream.Collectors.* to make the expressions easier to read.

You can also apply groupingBy twice:

```
Map<String, Map<String, List<Locale>>> countryAndLanguageToLoc
   Locale.availableLocales().collect(
      groupingBy(Locale::getCountry,
         groupingBy(Locale::getLanguage)));
```

Then countryAndLanguageToLocale.get("IN").get("hi") is a list of the Hindi locales in India. (There are several variants.)

Several collectors are provided for reducing collected elements to numbers:

- counting produces a count of the collected elements. For example,

  ```
  Map<String, Long> countryToLocaleCounts = Locale.availableL
     groupingBy(Locale::getCountry, counting()));
  ```

  counts how many locales there are for each country.
- summing(Int|Long|Double) and averaging(Int|Long|Double) apply a provided function to the downstream elements and produce the sum or average of the function's results. For example,

  ```
  public record City(String name, String state, int population) {}
  ```

```
. . .
Map<String, Integer> stateToCityPopulation = cities.collect(
   groupingBy(City::state, averagingInt(City::population)));
```

computes the average of populations per state in a stream of cities.

- maxBy and minBy take a comparator and produce maximum and minimum of the downstream elements. For example,

```
Map<String, Optional<City>> stateToLargestCity = cities.collect
   groupingBy(City::state,
      maxBy(Comparator.comparing(City::population))));
```

produces the largest city per state.

The collectingAndThen collector adds a final processing step behind a collector. For example, if you want to know how many distinct results there are, collect them into a set and then compute the size:

```
Map<Character, Integer> stringCountsByStartingLetter = strings.colle
   groupingBy(s -> s.charAt(0),
      collectingAndThen(toSet(), Set::size)));
```

The mapping collector does the opposite. It applies a function to each collected element and passes the results to a downstream collector.

```
Map<Character, Set<Integer>> stringLengthsByStartingLetter = strin
   groupingBy(s -> s.charAt(0),
      mapping(String::length, toSet())));
```

Here, we group strings by their first character. Within each group, we produce the lengths and collect them in a set.

The mapping method also yields a nicer solution to a problem from the preceding section—gathering a set of all languages in a country.

```
Map<String, Set<String>> countryToLanguages = Locale.availableLo
   groupingBy(Locale::getDisplayCountry,
```

```
        mapping(Locale::getDisplayLanguage,
          toSet())));
```

There is a flatMapping method as well, for use with functions that return streams.

If the grouping or mapping function has return type int, long, or double, you can collect elements into a summary statistics object, as discussed in Section 1.8. For example,

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary
  groupingBy(City::state,
    summarizingInt(City::population)));
```

Then you can get the sum, count, average, minimum, and maximum of the function values from the summary statistics objects of each group.

The filtering collector applies a filter to each group, for example:

```
Map<String, Set<City>> largeCitiesByState
  = cities.collect(
    groupingBy(City::state,
      filtering(c -> c.population() > 500000,
        toSet()))); // States without large cities have empty sets
```

Finally, you can use the teeing collector to branch into two downstream collections. This is useful whenever you need to compute more than one result from a stream. Suppose you want to collect city names and also compute their average population. You can't read a stream twice, but teeing lets you carry out two computations. Specify two downstream collectors and a function that combines the results.

```
record Pair<S, T>(S first, T second) {}
Pair<List<String>, Double> result = cities.filter(c -> c.state().equals("N
  .collect(teeing(
    mapping(City::name, toList()), // First downstream collector
    averagingDouble(City::population), // Second downstream collec
    (list, avg) -> new Pair(list,  avg))); // Combining function
```

> There are also three versions of a reducing method that apply general reductions, as described in the next section.

Composing collectors is powerful, but it can lead to very convoluted expressions. The best use is with groupingBy or partitioningBy to process the “downstream” map values. Otherwise, simply apply methods such as map, reduce, count, max, or min directly on streams.

The example program in Listing 1.7 demonstrates downstream collectors.

Listing 1.7 **v2ch01/collecting/DownstreamCollectors.java**

```
package v2ch01.collecting;

import static java.util.stream.Collectors.*;

import module java.base;

record City(String name, String state, int population) {}

class DownstreamCollectors {

   void main() throws Exception {
      Map<String, Set<Locale>> countryToLocaleSet = Locale.availableLocales()
            .collect(groupingBy(Locale::getCountry, toSet()));
      IO.println("countryToLocaleSet: " + countryToLocaleSet);

      Map<String, Map<String, List<Locale>>> countryAndLanguageToLocale =
            .availableLocales()
            .collect(groupingBy(Locale::getCountry, groupingBy(Locale::getLangu
      IO.println(
         "Hindi locales in India: " + countryAndLanguageToLocale.get("IN").get("

      Map<String, Long> countryToLocaleCounts = Locale.availableLocales()
            .collect(groupingBy(Locale::getCountry, counting()));
      IO.println("countryToLocaleCounts: " + countryToLocaleCounts);
```

```
      Stream<City> cities = readCities("v2ch01/cities.txt");
      Map<String, Integer> stateToCityPopulation = cities
            .collect(groupingBy(City::state, summingInt(City::population)));
      IO.println("stateToCityPopulation: " + stateToCityPopulation);

      cities = readCities("v2ch01/cities.txt");
      Map<String, Optional<String>> stateToLongestCityName = cities.collect(gr
         City::state, mapping(City::name, maxBy(Comparator.comparing(String::l
      IO.println("stateToLongestCityName: " + stateToLongestCityName);

      Map<String, Set<String>> countryToLanguages = Locale.availableLocales(
            .collect(groupingBy(Locale::getDisplayCountry,
               mapping(Locale::getDisplayLanguage, toSet())));
      IO.println("countryToLanguages: " + countryToLanguages);

      cities = readCities("v2ch01/cities.txt");
      Map<String, IntSummaryStatistics> stateToCityPopulationSummary = citie
            .collect(groupingBy(City::state, summarizingInt(City::population)));
      IO.println(stateToCityPopulationSummary.get("NY"));

      cities = readCities("v2ch01/cities.txt");
      Map<String, String> stateToCityNames = cities.collect(groupingBy(City::sta
         reducing("", City::name, (s, t) -> s.length() == 0 ? t : s + ", " + t)));

      cities = readCities("v2ch01/cities.txt");
      stateToCityNames = cities
            .collect(groupingBy(City::state, mapping(City::name, joining(", "))));
      IO.println("stateToCityNames: " + stateToCityNames);

      cities = readCities("v2ch01/cities.txt");
      record Pair<S, T>(S first, T second) {
      }
      Pair<List<String>, Double> result = cities.filter(c -> c.state().equals("NV")).c
         teeing(mapping(City::name, toList()), averagingDouble(City::population)
            (names, avg) -> new Pair<>(names, avg)));
      IO.println(result);
   }

   Stream<City> readCities(String filename) throws IOException {
      return Files.lines(Path.of(filename)).map(l -> l.split(", "))
            .map(a -> new City(a[0], a[1], Integer.parseInt(a[2])));
   }
}
```

java.util.stream.Collectors8

- public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)
  yields a collector that produces a map. The keys are the results of applying classifier to all collected elements. The values are the results of collecting elements with the same key, using the downstream collector.
- static <T> Collector<T,?,Long> counting()
  yields a collector that counts the collected elements.
- static <T> Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)
- static <T> Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)
- static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)
  yield a collector that computes the sum of the results of applying mapper to the collected elements.
- static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)
- static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)
  yield a collector that computes the maximum or minimum of the collected elements, using the ordering specified by comparator.
- static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)
  yields a collector that sends elements to the downstream collector and then applies the finisher function to its result.
- static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)
  yields a collector that calls mapper on each element and passes the results to the downstream collector.
- static <T,U,A,R> Collector<T,?,R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream)

yields a collector that calls mapper on each element and passes the elements of the results to the downstream collector.
- static <T,A,R> Collector<T,?,R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream) yields a collector that passes the elements fulfilling the predicate to the downstream collector.

1.9.5. Implementing Collectors

In the collection process, a collector accumulates incoming stream elements in an internal data structure called a "result container." Each collector can choose a suitable result container type. If the stream is parallel, multiple result containers are filled concurrently and then merged. After accumulation and merging, the collector can optionally transform the final result container into another object that is the collection result.

Specifically, a Collector must implement four methods, each of which yields a function object:

- Supplier<A> supplier() supplies a result container of type A.
- BiConsumer<A,T> accumulator() accumulates a stream element of type T into a result container.
- BinaryOperator<A> combiner() combines two result containers into one.
- Function<A, R> finisher() transforms the final result container into the result of type R.

Let's look at the function objects of the Collectors.toList() collector:

- The supplier is a function yielding an empty ArrayList<T> or simply ArrayList<T>::new.
- The accumulator is (rc, t) -> rc.add(t) or List::add. (That method expression is why the result container is the first parameter.)
- The combiner concatenates two lists: (rc1, rc2) -> {

rc1.addAll(rc2); return rc1; }.
- The finisher does nothing (which is the most common case).

The characteristics method of a collector returns a set of Collector.Characteristics flags that can help the stream with optimizing the collection process. There are three of them:

- IDENTITY_FINISH: the finisher need not be called.
- UNORDERED: it is ok not to call the accumulator in encounter order.
- CONCURRENT: it is safe to call the accumulator from multiple threads on the same result container.

Collectors.joining returns a collector with none of these characteristics. The result container is a StringBuilder. The finisher must convert it into a string. Accumulation order matters. And the result container is not threadsafe.

In contrast, the toConcurrentMap collector has all three characteristics. The result container is a threadsafe ConcurrentHashMap. Insertion order does not matter, and no finishing is needed.

> Collectors are meant to work efficiently with parallel streams, which makes them unsuitable for certain terminal operations. Consider collecting every *n*th element from a stream. That can't be parallelized, since the accumulator can't track the indexes in each segment. But a collector can’t tell the stream that it wants the elements sequentially.
>
> In this situation, you are better off using the gatherer API —see Section 1.11.

Let’s build a collector that statistically samples elements, accepting them with a given probability. For now, we’ll collect the accepted elements in a list.

Following the Collectors API, let’s make a method that yields the

collector:

```
<T> Collector<T, List<T>, List<T>> randomSampling(double p) {
  return Collector.of(ArrayList<T>::new,
        (resultContainer, e) -> {
          if (Math.random() < p)
            resultContainer.add(e);
          },
        (rc1, rc2) -> { rc1.addAll(rc2); return rc1; },
        Function.identity(),
        Collector.Characteristics.IDENTITY_FINISH);
  }
```

The Collector.of method has five parameters: the four functions and a varargs parameter for the characteristics.

This collector is almost the same as the toList collector, except that the accumulator only stores some of the elements.

Here is how to use it:

```
Stream<Integer> numbers = . . .;
List<Integer> sampled = numbers.collect(randomSampling(0.05)); // A
```

What if we don't want the accepted elements in a list? Our API should support downstream collectors. Here is how:

```
<T, A, R> Collector<T, A, R> randomSampling(double p, Collector<T, A
      BiConsumer<A, T> downstreamAccumulator = downstream.accu
      return Collector.of(downstream.supplier(),
            (resultContainer, e) -> {
              if (Math.random() < p)
                downstreamAccumulator.accept(resultContainer, e);
              },
            downstream.combiner(),
            downstream.finisher(),
            downstream.characteristics().toArray(Collector.Characterist
  }
```

Here, downstream is a collector that accepts elements of type T, with a result container of type A, and result type R.

We can reuse the downstream's supplier, combiner, and finisher, but we must adapt its accumulator, so that only a fraction of the incoming elements are inserted.

Unfortunately, the API leaves something to be desired for this particular use case. I have to turn the downstream's set of characteristics into an array for the varargs parameter.

The program in Listing 1.8 has the complete example.

Listing 1.8 **v2ch01/collecting/ImplementingCollectors.java**

```
package v2ch01.collecting;

import module java.base;

class ImplementingCollectors {
   void main() {
      IO.println(IntStream.range(1, 1000)
         .boxed()
         .parallel()
         .collect(randomSampling(0.05)));
      IO.println(IntStream.range(1, 1000)
         .boxed()
         .parallel()
         .collect(randomSampling(0.05, Collectors.toSet())));
   }

   <T> Collector<T, List<T>, List<T>> randomSampling(double p) {
      return Collector.of(ArrayList<T>::new,
            (resultContainer, e) -> {
               if (Math.random() < p)
                  resultContainer.add(e);
               },
            (rc1, rc2) -> { rc1.addAll(rc2); return rc1; },
            Function.identity(),
            Collector.Characteristics.IDENTITY_FINISH);
   }

   <T, A, R> Collector<T, A, R> randomSampling(double p, Collector<T, A, R> dow
      BiConsumer<A, T> downstreamAccumulator = downstream.accumulator(
      return Collector.of(downstream.supplier(),
```

```
            (resultContainer, e) -> {
               if (Math.random() < p)
                  downstreamAccumulator.accept(resultContainer, e);
               },
            downstream.combiner(),
            downstream.finisher(),
            downstream.characteristics().toArray(Collector.Characteristics[]::new
   }
}
```

java.util.stream.Collector<T, A, R>8

- Supplier<A> supplier()
  The function to supply a new result container.
- BiConsumer<A,T> accumulator()
  The function to incorporate an element from the stream into a result container.
- BinaryOperator<A> combiner()
  The function to combine two result containers.
- Function<A,R> finisher()
  The function that turns the final result container into the result of the collector
- Set<Collector.Characteristics> characteristics()
  The characteristics of this collector.
- static <T,A,R> Collector<T,A,R> of(Supplier<A> supplier, BiConsumer<A,T> accumulator, BinaryOperator<A> combiner, Function<A,R> finisher, Collector.Characteristics... characteristics)
  yields a collector with the given supplier, accumulator, combiner, finisher, and characteristics.

1.10. Reduction Operations

The reduce method is another general mechanism for computing a value from a stream. In particular with mathematical operations, it is much easier to specify a reduction than it would be to implement the same operation with a collector. The simplest form takes a binary function and keeps applying it, starting with the first two elements. It's easy

to explain this if the function is the sum:

```
List<Integer> values = . . .;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

In this case, the reduce method computes $v_0 + v_1 + v_2 + ...$, where v_i are the stream elements. The method returns an Optional because there is no valid result if the stream is empty.

> In this case, you can write reduce(Integer::sum) instead of reduce((x, y) -> x + y).

More generally, you can use any operation that combines a partial result x with the next value y to yield a new partial result.

Here is another way of looking at reductions. Given a reduction operation *op*, the reduction yields v_0 *op* v_1 *op* v_2 *op* ..., where v_i *op* v_{i+1} denotes the function call $op(v_i, v_{i+1})$. There are many operations that might be useful in practice—such as sum, product, string concatenation, maximum and minimum, set union or intersection.

If you want to use reduction with parallel streams, the operation must be associative: It shouldn't matter in which order you combine the elements. In math notation, $(x\ op\ y)\ op\ z$ must be equal to $x\ op\ (y\ op\ z)$. An example of an operation that is not associative is subtraction. For example, $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Often, there is an identity *e* such that $e\ op\ x = x$, and that element can be used as the start of the computation. For example, 0 is the identity for addition, and you can use the second form of reduce:

```
List<Integer> values = . . .;
Integer sum = values.stream().reduce(0, (x, y) -> x + y);
   // Computes 0 + v0 + v1 + v2 + . . .
```

The identity value is returned if the stream is empty, and you

no longer need to deal with the Optional class.

Now suppose you have a stream of objects and want to form the sum of some property, such as lengths in a stream of strings. You can't use the simple form of reduce. It requires a function (T, T) -> T, with the same types for the parameters and the result, but in this situation you have two types: The stream elements have type String, and the accumulated result is an integer. There is a form of reduce that can deal with this situation.

First, you supply an “accumulator” function (total, word) -> total + word.length(). That function is called repeatedly, forming the cumulative total. But when the computation is parallelized, there will be multiple computations of this kind, and you need to combine their results. You supply a second function for that purpose. The complete call is

```
int result = words.reduce(0,
   (total, word) -> total + word.length(),
   (total1, total2) -> total1 + total2);
```

In practice, you probably won't use the reduce method a lot. It is usually easier to map to a stream of numbers and use one of its methods to compute sum, maximum, or minimum. (We discuss streams of numbers in Section 1.12.) In this particular example, you could have called words.mapToInt(String::length).sum(), which is both simpler and more efficient since it doesn’t involve boxing.

There are times when reduce is not general enough. For example, suppose you want to collect the results in a BitSet. If the collection is parallelized, you can’t put the elements directly into a single BitSet because a BitSet object is not thread-safe. For that reason, you can't use reduce. Each segment needs to start out with its own empty set, and reduce only lets you supply one identity value. Instead, use the three-parameter form of collect.

You provide three functions:

1. A supplier to construct new instances of the target object
2. An accumulator that adds an element to the target
3. A combiner that merges two target objects into one

Here is how the collect method works for a bit set:

```
BitSet result = stream.collect(BitSet::new, BitSet::set, BitSet::or)
```

java.util.Stream<T>8

- Optional<T> reduce(BinaryOperator<T> accumulator)
- T reduce(T identity, BinaryOperator<T> accumulator)
- <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
  form a cumulative total of the stream elements with the given accumulator function. If identity is provided, then it is the first value to be accumulated. If combiner is provided, it can be used to combine totals of segments that are accumulated separately.
- <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
  collects elements in a result of type R. On each segment, supplier is called to provide an initial result, accumulator is called to mutably add elements to it, and combiner is called to combine two results.

1.11. Gatherers

The stream API has an extension point for implementing arbitrary terminal operations: the collect method, whose argument is a Collector. Many collector implementations are provided in the Collectors class. You can also implement your own, as you have seen in Section 1.9.5.

Java 24 adds an extension point for intermediate operations.

The gather method, whose argument is a Gatherer, turns a stream into another stream.

1.11.1. Predefined Gatherers

The Gatherers class provides a few Gatherer implementations. The windowFixed and windowSliding gatherers group adjacent elements together, yielding a stream of lists. It's easier to show them in action than to explain them in words:

```
IntStream.range(0, 10).boxed().gather(Gatherers.windowFixed(4)).to
   // [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9]]
IntStream.range(0, 10).boxed().gather(Gatherers.windowSliding(4)).t
   // [[0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6],
   // [4, 5, 6, 7], [5, 6, 7, 8], [6, 7, 8, 9]]
```

The windowSliding gatherer is useful for comparing adjacent elements of a stream. For example, here is how to drop adjacent duplicates:

```
Stream.concat(words, Stream.of((String) null))
   .gather(Gatherers.windowSliding(2))
   .filter(w -> !w.get(0).equalsIgnoreCase(w.get(1)))
   .map(w -> w.get(0))
```

By appending a null to the stream, the last element can be compared with something.

The mapConcurrent gatherer is similar to map, but the computations run in virtual threads. This is very useful for blocking operations. For example, if you have a list of URLs, you can read them concurrently.

```
String read(String url) {
   try {
      return new String(URI.create(url).toURL().openStream().readAll
   }
   catch (IOException e) {
      throw new UncheckedIOException(e);
   }
}
```

```
int max = 100; // Maximumum number of concurrent threads
List<String> contents = urls.stream()
   .gather(Gatherers.mapConcurrent(max, this::read))
   .toList();
```

If one of the computations throws an exception, all remaining tasks are cancelled, and the exception is rethrown.

> As you will see in Section 1.13, parallel streams are used to parallelize computationally intensive operations over multiple processor cores. In contrast, the mapConcurrent gatherer is intended for concurrent operations that mostly block.

Finally, there are fold and scan gatherers. A fold is a general mechanism for repeatedly applying an operation *op*. It starts with an initial value *iv* and computes (((*iv op* v_0) *op* v_1) *op* v_2 *op*) ..., where v_i are the stream elements.

This looks similar to a reduction, which you saw in Section 1.10. However, reductions are intended for parallelizable operations. A fold is inherently sequential, and the operation need not be associative.

Some functional programmers love folds because any loop can be translated into a fold. Consider this computation of a number from its decimal digits:

```
int n = 0;
for (int d : digits)
   n = 10 * n + d;
```

Here is the same computation as a fold:

```
int n = digits.stream()
   .gather(Gatherers.fold(() -> 0, (x, y) -> x * 10 + y))
   .findFirst()
   .orElse(0);
```

The fold gatherer produces a stream with one element, or none if digits was empty.

As you can see from the example, you have to provide a supplier for the initial value, and the operation that is repeatedly applied.

> It seems odd that folding is implemented as a gatherer and not a collector. It would be more natural as a terminal operation since it produces a single result. But collectors are unable to signal that they must receive stream elements sequentially. As you will see in Section 1.11.2, gatherers have this capability. But a gatherer must yield a stream; in this case, one of length zero or one.

There is also a scan gatherer that produces a stream of all intermediate results. For example,

```
Stream.of(1, 7, 2, 9)
   .gather(Gatherers.scan(() -> 0, (x, y) -> x * 10 + y))
   .toList() // [1, 17, 172, 1729]
```

1.11.2. Implementing Gatherers

Implementing a gatherer is similar to implementing a collector (Section 1.9.5). A gatherer has a generic "intermediate state" object instead of the "result collection" of a collector. It too has four methods yielding function objects. Here T is the type of the stream elements, A the type of the intermediate state, and R the element type of the stream that the gatherer produces.

- Supplier<A> initializer() supplies an intermediate state instance.
- Gatherer.Integrator<A, T, R> integrator() processes a stream element.
- BinaryOperator<A> combiner() combines two intermediate states int one.
- BiConsumer<A, Gatherer.Downstream<? super R>> finisher() carries out a final operation after processing all

stream elements.

Gatherer.Integrator is a functional interface with this method:

```
boolean integrate(A state, T element, Gatherer.Downstream<? super
```

In the integrator and finisher, you call the push method of the Gatherer.Downstream interface in order to send values to the stream that the gatherer produces. You can call push as often as you like, or not at all.

Very simple gatherers can be stateless. Then the initializer is not needed.

A combiner is only needed for parallel streams.

The finisher is only needed if you need to push values after the last element has been integrated. For example, the windowFixed gatherer pushes the short list of remaining elements in the finisher.

As an example, let us implement a gatherer that samples every *n*th element of a stream.

Here is how we want to use it:

```
List<String> samples = wordList.stream().gather(sampling(50)).toLis
```

The result is a list holding every 50th word from the original list.

The sampling method yields the gatherer:

```
static class IntermediateState {
   long index = -1;
}

<T> Gatherer<T, IntermediateState, T> sampling(int n) {
   return Gatherer.ofSequential(
      IntermediateState::new,
      Gatherer.Integrator.of((state, e, downstream) -> {
         state.index++;
```

```
        if (state.index % n == 0)
            return downstream.push(e);
        else
            return true;
        }));
    }
```

The Gatherer.ofSequential method yields a gatherer that calls the integrator in encounter order. There is also a Gatherer.of method for constructing parallelizable gatherers.

This gatherer doesn't have a combiner or finisher. In general, you don’t provide arguments for unneeded initializers, combiners and finishers. Use one of the overloaded versions with just the gatherer operations you need.

> Use these factory methods rather than declaring an explicit class implementing the Gatherer interface. It is less work, and it is easier to tell whether your gatherer is sequential or parallelizable.
>
> You may wonder how the stream implementation distinguishes the two. A gatherer is sequential if the combiner method yields the exact same do-nothing instance as the one returned by the static defaultCombiner method. The factory methods handle this detail.
>
> If you declare your own class, you need to be careful. A sequential gatherer should not override combiner but instead keep the default method, which returns defaultCombiner(). If you have a parallelizable gatherer that doesn’t need a combiner, you need to declare one anyway.

Now let us look more closely at the sampling gatherer. The state holds the index of the element. The initializer constructs a new state instance.

The integrator is declared using the Gatherer.Integrator.of

factory method. Whenever an element is processed, the index is incremented. The element is pushed when the index is divisible by the sampling count.

Note that the integrator returns a boolean. You should return false when the pipeline “short-circuits” and no longer wants to accept elements. This can happen for two reasons. Your gatherer might want to short-circuit on its own. Simply return false in that case. (In the sampling example, that never happens.) Alternatively, a downstream operation can short-circuit. Then the call to push returns false. The integrator needs to return that downstream status.

> The Gatherer.Downstream interface has a method isRejecting. It allows an integrator to check whether a value would be accepted before it embarks on a costly computation.
>
> Do not use the isRejecting method for reporting the downstream status. You want to keep the integrator code as simple as possible. Then it is more likely that the just-in-time compiler will inline it.

For optimizing the performance of a stream pipeline, it is important to know whether it contains any short-circuiting operations. An integrator that doesn't short-circuit should signal that fact by implementing the Gatherer.Integrator.Greedy interface. There is a factory method for that:

```
<T> Gatherer<T, IntermediateState, T> sampling(int n) {
   return Gatherer.ofSequential(
      IntermediateState::new,
      Gatherer.Integrator.ofGreedy((state, e, downstream) -> {
         state.index++;
         if (state.index % n == 0)
            return downstream.push(e);
         else
            return true;
         }));
}
```

You could create a class that implements the interface, but using a factory method is easier.

The Gatherer.Integrator.of method that you saw in the first implementation was not strictly necessary. We could have just used a lambda expression. It would have been converted to a Gatherer.Integrator. But the factory method tells the reader more clearly that this is not a greedy integrator.

As you have seen, writing your own gatherer may seem a bit daunting at first, but it is not all that complex. Follow these steps:

1. Decide whether your gatherer can execute in parallel (Gatherer.of), or whether it must see the elements sequentially (Gatherer.ofSequential).
2. Decide whether the integrator may short-circuit (Gatherer.Integrator.of) or if it always accepts elements (Gatherer.Integrator.ofGreedy).
3. Do you need state? If so, provide a class for the state, and if your gatherer is not sequential, implement a combiner.
4. In the integrator, return false when short-ciruiting, or when the downstream push is rejected.

Unlike collectors, gatherers don't use enumerations for their characteristics. They use these alternative mechanisms:

1. The integrator of a greedy gatherer implements Gatherer.Integrator.Greedy. Otherwise the gatherer is short-circuiting.
2. The combiner of a sequential gatherer is the object returned by Gatherer.defaultCombiner(). Otherwise the gatherer is parallelizable.
3. The initializer of a stateless gatherer is the object returned by Gatherer.defaultInitializer(). Otherwise

the gatherer is stateful.

java.util.Stream<T>8

- <R> Stream<R> gather(Gatherer<? super T, ?, R> gatherer)24
  yields a stream that results from processing the elements of this stream with the given gatherer.

java.util.stream.Gatherers24

- static <TR> Gatherer<TR,?,List<TR>> windowFixed(int windowSize)
  static <TR> Gatherer<TR,?,List<TR>> windowSliding(int windowSize)
  yields a gatherer that groups stream elements into separate or overlapping windows of the given size.
- static <T,R> Gatherer<T,?,R> fold(Supplier<R> initial, BiFunction<? super R, ? super T, ? extends R> folder)
  static <T,R> Gatherer<T,?,R> scan(Supplier<R> initial, BiFunction<? super R, ? super T, ? extends R> scanner)
  yield a gatherer that repeatedly applies the binary operation to the preceding result and the next stream element. fold produces only the final result, while scan yields all intermediate results.
- static <T,R> Gatherer<T,?,R> mapConcurrent(int maxConcurrency, Function<? super T, ? extends R> mapper)
  yields a gatherer that applies the given function, running the computations in at most maxConcurrency virtual threads, producing a stream of the results.

java.util.stream.Gatherer.Downstream<T>24

- boolean push(T element)
  pushes the element to the next stage of the processing

pipeline. Returns false if the element was rejected.

- boolean isRejecting()
  returns true if the processing pipeline no longer accepts elements.

java.util.stream.Gatherer<T, A, R>[24]

- Supplier<A> initializer()
  Gatherer.Integrator<A,T,R> integrator()
  BinaryOperator<A> combiner()
  BiConsumer<A, Gatherer.Downstream<? super R>> finisher()
  return the initializer, integrator, combiner, and finisher of this gatherer.
- static <T,R> Gatherer<T,Void,R> of(Gatherer.Integrator<Void,T,R> integrator)
  static <T,R> Gatherer<T,Void,R> of(Gatherer.Integrator<Void,T,R> integrator, BiConsumer<Void, Gatherer.Downstream<? super R>> finisher)
  static <T,A,R> Gatherer<T,A,R> of(Supplier<A> initializer, Gatherer.Integrator<A,T,R> integrator, BinaryOperator<A> combiner, BiConsumer<A, Gatherer.Downstream<? super R>> finisher)
  yield a parallelizable gatherer with the given initializer, integrator, combiner, and finisher. When there is no initializer, the gatherer is stateless.
- static <T,R> Gatherer<T,Void,R> ofSequential(Gatherer.Integrator<Void,T,R> integrator)
  static <T,A,R> Gatherer<T,A,R> ofSequential(Supplier<A> initializer, Gatherer.Integrator<A,T,R> integrator)
  static <T,R> Gatherer<T,Void,R> ofSequential(Gatherer.Integrator<Void,T,R> integrator, BiConsumer<Void, Gatherer.Downstream<? super R>> finisher)
  static <T,A,R> Gatherer<T,A,R> ofSequential(Supplier<A> initializer, Gatherer.Integrator<A,T,R> integrator, BiConsumer<A, Gatherer.Downstream<? super R>> finisher)

yield a sequential gatherer with the given initializer, integrator, and finisher. When there is no initializer, the gatherer is stateless.

java.util.stream.Gatherer.Integrator<T>[24]

- boolean integrate(A state, T element, Gatherer.Downstream<? super R> downstream)
  This method is called for each stream element. Update the state, and push a result downstream when appropriate. Return false if further elements will be rejected.
- static <A,T,R> Gatherer.Integrator<A,T,R> of(Gatherer.Integrator<A,T,R> integrator)
  static <A,T,R> Gatherer.Integrator.Greedy<A,T,R> ofGreedy(Gatherer.Integrator.Greedy<A,T,R> greedy)
  yield a short-circuiting or greedy integrator. The argument is intended to be a lambda expression.

1.12. Primitive Type Streams

So far, we have collected integers in a Stream<Integer>, even though it is clearly inefficient to wrap each integer into a wrapper object. The same is true for the other primitive types—double, float, long, short, char, byte, and boolean. The stream library has specialized types IntStream, LongStream, and DoubleStream that store primitive values directly, without using wrappers. If you want to store short, char, byte, and boolean, use an IntStream; for float, use a DoubleStream.

To create an IntStream, call the IntStream.of and Arrays.stream methods:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to); // values is an int[] array
```

As with object streams, you can also use the static generate and iterate methods. In addition, IntStream and LongStream have

static methods range and rangeClosed that generate integer ranges with step size one:

```
IntStream zeroToNinetyNine = IntStream.range(0, 100); // Upper bou
IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // Upper
```

The CharSequence interface has methods codePoints and chars that yield an IntStream of the Unicode codes of the characters or of the code units in the UTF-16 encoding. (See Chapter 2 for the sordid details.)

```
String greeting = "Ahoy ☠";
IntStream codes = greeting.codePoints();
   // The stream with values 67, 105, 97, 111, 32, 127988, 8205, 9760, 6
```

The RandomGenerator interface has methods ints, longs, and doubles that return primitive type streams of random numbers.

```
IntStream randomIntegers = RandomGenerator.getDefault().ints();
```

When you have a stream of objects, you can transform it to a primitive type stream with the mapToInt, mapToLong, or mapToDouble methods. For example, if you have a stream of strings and want to process their lengths as integers, you might as well do it in an IntStream:

```
Stream<String> words = . . .;
IntStream lengths = words.mapToInt(String::length);
```

When converting a Stream<Integer> to an IntStream, use

```
numbers.mapToInt(i -> i)
```

and not

```
numbers.mapToInt(Function.identity()) // ERROR
```

The latter is a compile-time error, since there is no automatic unboxing.

To convert a primitive type stream to an object stream, use the boxed method:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Generally, the methods on primitive type streams are analogous to those on object streams. Here are the most notable differences:

- The toArray methods return primitive type arrays.
- Methods that yield an optional result return an OptionalInt, OptionalLong, or OptionalDouble. These classes are analogous to the Optional class, but they have methods getAsInt, getAsLong, and getAsDouble instead of the get method.
- There are methods sum, average, max, and min that return the sum, count, average, maximum, and minimum. These methods are not defined for object streams.
- The summaryStatistics method yields an object of type IntSummaryStatistics, LongSummaryStatistics, or DoubleSummaryStatistics that can simultaneously report the sum, count, average, maximum, and minimum of the stream.

The program in Listing 1.9 gives examples for the API of primitive type streams.

Listing 1.9 **v2ch01/streams/PrimitiveTypeStreams.java**

```
package v2ch01.streams;

import module java.base;

class PrimitiveTypeStreams {
   void main() throws Exception {
      IntStream is1 = IntStream.generate(() -> (int) (Math.random() * 100));
      show("is1", is1);
      IntStream is2 = IntStream.range(5, 10);
      show("is2", is2);
```

```
      IntStream is3 = IntStream.rangeClosed(5, 10);
      show("is3", is3);

      String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));

      Stream<String> words = Stream.of(contents.split("\\PL+"));
      IntStream is4 = words.mapToInt(String::length);
      show("is4", is4);
      String sentence = "\uD835\uDD46 is the set of octonions.";
      IO.println(sentence);
      IntStream codes = sentence.codePoints();
      IO.println(codes.mapToObj(c -> "%X ".formatted(c)).collect(Collectors.joini

      Stream<Integer> integers = IntStream.range(0, 100).boxed();
      IntStream is5 = integers.mapToInt(Integer::intValue);
      show("is5", is5);
   }

   void show(String title, IntStream stream) {
      final int SIZE = 10;
      int[] firstElements = stream.limit(SIZE + 1).toArray();
      IO.print(title + ": ");
      for (int i = 0; i < firstElements.length; i++) {
         if (i > 0) IO.print(", ");
         if (i < SIZE)
            IO.print(firstElements[i]);
         else
            IO.print("...");
      }
      IO.println();
   }
}
```

java.util.stream.IntStream8

- static IntStream range(int startInclusive, int endExclusive)
- static IntStream rangeClosed(int startInclusive, int endInclusive)
  yield an IntStream with the integers in the given range.
- static IntStream of(int... values)
  yields an IntStream with the given elements.
- int[] toArray()
  yields an array with the elements of this stream.

- int sum()
- OptionalDouble average()
- OptionalInt max()
- OptionalInt min()
- IntSummaryStatistics summaryStatistics()
  yield the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- Stream<Integer> boxed()
  yields a stream of wrapper objects for the elements in this stream.

java.util.stream.LongStream8

- static LongStream range(long startInclusive, long endExclusive)
- static LongStream rangeClosed(long startInclusive, long endInclusive)
  yield a LongStream with the integers in the given range.
- static LongStream of(long... values)
  yields a LongStream with the given elements.
- long[] toArray()
  yields an array with the elements of this stream.
- long sum()
- OptionalDouble average()
- OptionalLong max()
- OptionalLong min()
- LongSummaryStatistics summaryStatistics()
  yield the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- Stream<Long> boxed()
  yields a stream of wrapper objects for the elements in this stream.

java.util.stream.DoubleStream8

- static DoubleStream of(double... values)
  yields a DoubleStream with the given elements.
- double[] toArray()
  yields an array with the elements of this stream.
- double sum()
- OptionalDouble average()
- OptionalDouble max()
- OptionalDouble min()
- DoubleSummaryStatistics summaryStatistics()
  yield the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- Stream<Double> boxed()
  yields a stream of wrapper objects for the elements in this stream.

java.lang.CharSequence1.0

- IntStream codePoints()8
  yields a stream of all Unicode code points.

java.util.random.RandomGenerator17

- IntStream ints()
- IntStream ints(int randomNumberOrigin, int randomNumberBound)
- IntStream ints(long streamSize)
- IntStream ints(long streamSize, int randomNumberOrigin, int randomNumberBound)
- LongStream longs()
- LongStream longs(long randomNumberOrigin, long randomNumberBound)
- LongStream longs(long streamSize)
- LongStream longs(long streamSize, long randomNumberOrigin, long randomNumberBound)
- DoubleStream doubles()
- DoubleStream doubles(double randomNumberOrigin,

double randomNumberBound)
- DoubleStream doubles(long streamSize)
- DoubleStream doubles(long streamSize, double randomNumberOrigin, double randomNumberBound)
yield streams of random numbers. If streamSize is provided, the stream is finite with the given number of elements. When bounds are provided, the elements are between randomNumberOrigin (inclusive) and randomNumberBound (exclusive).
- DoubleStream equiDoubles(double left, double right, boolean isLeftIncluded, boolean isRightIncluded)[22]
yields a stream of floating-point numbers between left and right, uniformly drawn from the largest possible sequence of equidistant numbers in the target interval. In contrast, the doubles methods draw uniformly from floating-point numbers in the interval between 0 and 1, then transforming into the target interval. That transformation can introduce a slight distortion.

java.util.OptionalInt[8]

java.util.OptionalLong[8]

java.util.OptionalDouble[8]

- static Optional*Xxx* of(*xxx* value)
yields an optional object with the supplied primitive type value.
- *xxx* getAs*Xxx*()
yields the value of this optional object, or throws a NoSuchElementException if it is empty.
- *xxx* orElse(*xxx* other)
- *xxx* orElseGet(*Xxx*Supplier other)
yield the value of this optional object, or the alternative value if this object is empty.
- void ifPresent(*Xxx*Consumer consumer)
If this optional object is not empty, passes its value to consumer.

java.util.IntSummaryStatistics8

java.util.LongSummaryStatistics8

java.util.DoubleSummaryStatistics8

- long getCount()
- *xxx* getSum()
- double getAverage()
- *xxx* getMax()
- *xxx* getMin()

  yield the count, sum, average, maximum, and minimum of the collected elements.

1.13. Parallel Streams

Streams make it easy to parallelize bulk operations. The process is mostly automatic, but you need to follow a few rules. First of all, you must have a parallel stream. You can get a parallel stream from any collection with the Collection.parallelStream() method:

```
Stream<String> parallelWords = words.parallelStream();
```

Moreover, the parallel method converts any sequential stream into a parallel one.

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

As long as the stream is in parallel mode when the terminal method executes, all intermediate stream operations will be parallelized.

When stream operations run in parallel, the intent is that the same result is returned as if they had run serially. It is important that the operations are stateless and can be executed in an arbitrary order.

Here is an example of something you cannot do. Suppose you want to count all short words in a stream of strings:

```
var shortWords = new int[12];
words.parallelStream().forEach(
   s -> { if (s.length() < 12) shortWords[s.length()]++; });
      // ERROR--race condition!
IO.println(Arrays.toString(shortWords));
```

This is very, very bad code. The function passed to forEach runs concurrently in multiple threads, each updating a shared array. As you saw in Chapter 10 of Volume I, that's a classic race condition. If you run this program multiple times, you are quite likely to get a different sequence of counts in each run—each of them wrong.

It is your responsibility to ensure that any functions you pass to parallel stream operations are safe to execute in parallel. The best way to do that is to stay away from mutable state. In this example, you can safely parallelize the computation if you group strings by length and count them:

```
Map<Integer, Long> shortWordCounts
   = words.parallelStream()
      .filter(s -> s.length() < 12)
      .collect(groupingBy(
         String::length,
         counting()));
```

By default, streams that arise from ordered collections (arrays and lists), from ranges, generators, and iterators, or from calling Stream.sorted, are ordered. Results are accumulated in the order of the original elements, and are entirely predictable. If you run the same operations twice, you will get exactly the same results.

Ordering does not preclude efficient parallelization. For example, when computing stream.map(fun), the stream can be partitioned into *n* segments, each of which is concurrently processed. Then the results are reassembled in order.

Some operations can be more effectively parallelized when the ordering requirement is dropped. By calling the Stream.unordered method, you indicate that you are not interested in ordering. One operation that can benefit from this is Stream.distinct. On an ordered stream, distinct retains the first of all equal elements. That impedes parallelization—the thread processing a segment can't know which elements to discard until the preceding segment has been processed. If it is acceptable to retain any of the unique elements, all segments can be processed concurrently (using a shared set to track duplicates).

You can also speed up the limit method by dropping ordering. If you just want any n elements from a stream and you don’t care which ones you get, call

```
Stream<String> sample = words.parallelStream().unordered().limit(n
```

As discussed in Section 1.9.2, merging maps is expensive. For that reason, the Collectors.groupingByConcurrent method uses a shared concurrent map. To benefit from parallelism, the order of the map values will not be the same as the stream order.

```
Map<Integer, List<String>> result = words.parallelStream().collect(
   Collectors.groupingByConcurrent(String::length));
   // Values aren't collected in stream order
```

Of course, you won't care if you use a downstream collector that is independent of the ordering, such as

```
Map<Integer, Long> wordCounts
   = words.parallelStream()
      .collect(
         groupingByConcurrent(
            String::length,
            counting()));
```

Don't turn all your streams into parallel streams in the hope of speeding up operations. Keep these issues in mind:

- There is a substantial overhead to parallelization that will only pay off for very large data sets.
- Parallelizing a stream is only a win if the underlying data source can be effectively split into multiple parts.
- The thread pool that is used by parallel streams can be starved by blocking operations such as file I/O or network access.

Parallel streams work best with huge in-memory collections of data and computationally intensive processing.

If you process the lines of a huge file, parallelizing the stream may improve performance. The Files.lines method uses a memory-mapped file, to make splitting effective. (If you simply read a line at a time, then you would have to read the first half of the file before the second half, and parallelization would not work.)

If you need random numbers in parallel streams, obtain a generator from the RandomGenerator.SplittableGenerator.of factory method.

By default, parallel streams use the global fork-join pool returned by ForkJoinPool.commonPool. That is fine if your operations don't block and you don't share the pool with other tasks. There is a trick to substitute a different fork-join pool. Execute the parallel stream operations in a thread from that pool:

```
ForkJoinPool customPool = . . .;
result = customPool.submit(() ->
   stream.parallel().map(. . .).collect(. . .)).get();
```

Or, asynchronously:

```
CompletableFuture.supplyAsync(() ->
      stream.parallel().map(. . .).collect(. . .), customPool)
   .thenAccept(result -> . . .);
```

This only works with a fork-join pool, not with arbitrary executors.

The example program in Listing 1.10 demonstrates how to work with parallel streams.

Listing 1.10 **v2ch01/parallel/ParallelStreams.java**

```
package v2ch01.parallel;

import static java.util.stream.Collectors.*;

import module java.base;

class ParallelStreams {
   void main() throws Exception {
      String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
      List<String> wordList = List.of(contents.split("\\PL+"));

      // Very bad code ahead
      var shortWords = new int[10];
      wordList.parallelStream().forEach(s -> {
         if (s.length() < 10) shortWords[s.length()]++;
      });
      IO.println(Arrays.toString(shortWords));

      // Try again--the result will likely be different (and also wrong)
      Arrays.fill(shortWords, 0);
      wordList.parallelStream().forEach(s -> {
         if (s.length() < 10) shortWords[s.length()]++;
      });
      IO.println(Arrays.toString(shortWords));

      // Remedy: Group and count
      Map<Integer, Long> shortWordCounts = wordList.parallelStream()
            .filter(s -> s.length() < 10).collect(groupingBy(String::length, counting()

      IO.println(shortWordCounts);

      // Downstream order not deterministic
      Map<Integer, List<String>> result = wordList.parallelStream()
            .collect(Collectors.groupingByConcurrent(String::length));
```

```
        IO.println(result.get(14));

        result = wordList.parallelStream()
              .collect(Collectors.groupingByConcurrent(String::length));

        IO.println(result.get(14));

        Map<Integer, Long> wordCounts = wordList.parallelStream()
              .collect(groupingByConcurrent(String::length, counting()));

        IO.println(wordCounts);
    }
}
```

java.util.stream.BaseStream<T,S extends BaseStream<T,S>>**8**

- S parallel()
  yields a parallel stream with the same elements as this stream.
- S unordered()
  yields an unordered stream with the same elements as this stream.

java.util.Collection<E>**1.2**

- Stream<E> parallelStream()**8**
  yields a parallel stream with the elements of this collection.

In this chapter, you have learned how to put the stream library to use. The next chapter covers another important topic: processing input and output.

Chapter 2 ▪ Input and Output

In this chapter, we cover the Java Application Programming Interfaces (APIs) for input and output. You will learn how to access files and directories and how to read and write data in binary and text format. This chapter also shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data. We finish the chapter with a discussion of regular expressions, even though they are not actually related to input and output. We couldn't find a better place to handle that topic, and apparently neither could the Java team—the regular expression API specification was attached to a specification request for "new I/O" features.

2.1. Input/Output Streams

In the Java API, an object from which we can read a sequence of bytes is called an input stream. An object to which we can write a sequence of bytes is called an output stream. These sources and destinations of byte sequences can be—and often are—files, but they can also be network connections and even blocks of memory. The abstract classes InputStream and OutputStream are the basis for a hierarchy of input/output (I/O) classes.

> These input/output streams are unrelated to the streams that you saw in the preceding chapter. For clarity, we will use the terms input stream, output stream, or input/output stream whenever we discuss streams that are used for input and output.

Byte-oriented input/output streams are inconvenient for processing information stored in Unicode (recall that Unicode uses multiple bytes per character). Therefore, a separate hierarchy provides classes, inheriting from the abstract Reader and Writer classes, for processing Unicode characters. These classes have read and write operations that are based on two-byte char values (that is, UTF-16 code units) rather than byte values.

2.1.1. Reading and Writing Bytes

The InputStream class has an abstract method:

```
abstract int read()
```

This method reads one byte and returns the byte that was read, or -1 if it encounters the end of the input source. The designer of a concrete input stream class overrides this method to provide useful functionality. For example, in the FileInputStream class, this method reads one byte from a file. System.in is a predefined object of a subclass of InputStream that allows you to read information from “standard input,” that is, the console or a redirected file.

The InputStream has a very useful method to read all bytes of a stream:

```
byte[] bytes = in.readAllBytes();
```

There are also methods to read a given number of bytes—see the API notes.

These methods call the abstract read method, so subclasses need to override only one method.

Similarly, the OutputStream class defines the abstract method

```
abstract void write(int b)
```

which writes one byte to an output location.

If you have an array of bytes, you can write them all at once:

```
byte[] values = . . .;
out.write(values);
```

The transferTo method transfers all bytes from an input stream to an output stream:

```
in.transferTo(out);
```

Both the read and write methods block until the byte is actually read or written. This means that if the input stream cannot immediately be accessed (usually because of a busy network connection), the current thread blocks. This gives other threads the chance to do useful work while the method is waiting for the input stream to become available again.

The available method lets you check the number of bytes that are currently available for reading. This means a fragment like the following is unlikely to block:

```
int bytesAvailable = in.available();
if (bytesAvailable > 0) {
   var data = new byte[bytesAvailable];
   in.read(data);
}
```

When you have finished reading or writing to an input/output stream, close it by calling the close method. This call frees up the operating system resources that are in limited supply. If an application opens too many input/output streams without closing them, system resources can become depleted. Closing an output stream also flushes the buffer used for the output stream: Any bytes that were temporarily placed in a buffer so that they could be delivered as a larger packet are sent off. In particular, if you do not close a file, the last packet of bytes

might never be delivered. You can also manually flush the output with the flush method.

Even if an input/output stream class provides concrete methods to work with the raw read and write functions, application programmers rarely use them. The data that you are interested in probably contain numbers, strings, and objects, not raw bytes.

Instead of working with bytes, you can use one of many input/output classes that build upon the basic InputStream and OutputStream classes.

java.io.InputStream1.0

- abstract int read()
  reads a byte of data and returns the byte read; returns -1 at the end of the input stream.
- int read(byte[] b)
  reads into an array of bytes and returns the actual number of bytes read, or -1 at the end of the input stream; this method reads at most b.length bytes.
- int read(byte[] b, int off, int len)
- int readNBytes(byte[] b, int off, int len)9
  read up to len bytes, if available without blocking (read), or blocking until all values have been read (readNBytes). Values are placed into b, starting at off. Returns the actual number of bytes read.
- byte[] readAllBytes()9
  yields an array of all bytes that can be read from this stream.
- long transferTo(OutputStream out)9
  transfers all bytes from this input stream to the given output stream, returning the number of bytes transferred. Neither stream is closed.
- long skip(long n)
  attempts to skip n bytes in the input stream, returns the actual number of bytes skipped (which, for any reason, may be less than n).
- long skipNBytes(long n)12

skips n bytes in the input stream, returns the actual number of bytes skipped (which may be less than n if the end of the input stream was encountered).

- int available()
  returns the number of bytes available, without blocking (recall that blocking means that the current thread loses its turn).
- void close()
  closes the input stream.
- void mark(int readlimit)
  puts a marker at the current position in the input stream (not all streams support this feature). If more than readlimit bytes have been read from the input stream, the stream is allowed to forget the marker.
- void reset()
  returns to the last marker. Subsequent calls to read reread the bytes. If there is no current marker, the input stream is not reset.
- boolean markSupported()
  returns true if the input stream supports marking.
- static InputStream nullInputStream()11
  returns an input stream with no bytes.

java.io.OutputStream1.0

- abstract void write(int n)
  writes a byte of data.
- void write(byte[] b)
- void write(byte[] b, int off, int len)
  write all bytes, or len bytes starting at off, in the array b.
- void close()
  flushes and closes the output stream.
- void flush()
  flushes the output stream—that is, sends any buffered data to its destination.
- static OutputStream nullOutputStream()11
  returns an output stream that discards all bytes.

2.1.2. The Complete Stream Zoo

Unlike C, which gets by just fine with a single type FILE*, Java has a whole zoo of more than 60 (!) classes and interfaces for input and output.

.

Figure 2.1: Input and output stream hierarchy

.

Figure 2.2: Reader and writer hierarchy

Let's divide the animals in the input/output zoo by how they are used. There are separate hierarchies for classes that process bytes and characters. As you saw, the InputStream and OutputStream classes let you read and write individual bytes and arrays of bytes. These classes form the basis of the hierarchy shown in #ch01fig01. To read and write strings and numbers, you need more capable subclasses. For example, DataInputStream and DataOutputStream let you read and write all the primitive Java types in binary format. Finally, there are input/output streams for specialized tasks; for example, the ZipInputStream and ZipOutputStream let you read and write files in the familiar ZIP compression format.

For Unicode text, on the other hand, you can use subclasses of the abstract classes Reader and Writer (see #ch01fig02). The basic methods of the Reader and Writer classes are similar to those of InputStream and OutputStream.

```
abstract int read()
abstract void write(int c)
```

The read method returns either a UTF-16 code unit (as an integer between 0 and 65535) or -1 when you have reached the end of the file. The write method is called with a Unicode code unit. (See Chapter 3 of Volume I for a discussion of Unicode code units.)

There are four additional interfaces: Closeable, Flushable, Readable, and Appendable (see #ch01fig03). The first two interfaces are very simple, with methods

```
void close() throws IOException
```

and

```
void flush()
```

respectively. The classes InputStream, OutputStream, Reader, and Writer all implement the Closeable interface.

> The java.io.Closeable interface extends the java.lang.AutoCloseable interface. Therefore, you can use the try-with-resources statement with any Closeable. Why have two interfaces? The close method of the Closeable interface only throws an IOException, whereas the AutoCloseable.close method may throw any exception.

OutputStream and Writer implement the Flushable interface.

.

Figure 2.3: The Closeable, Flushable, Readable, and Appendable interfaces

The Readable interface has a single method

```
int read(CharBuffer cb)
```

The CharBuffer class has methods for sequential and random read/write access. It represents an in-memory buffer or a memory-mapped file. (See Section 2.4.2 for details.)

The Appendable interface has two methods for appending single characters and character sequences:

```
Appendable append(char c)
```

Appendable append(CharSequence s)

The CharSequence interface describes basic properties of a sequence of char values. It is implemented by String, CharBuffer, StringBuilder, and StringBuffer.

Of the input/output stream classes, only Writer implements Appendable.

java.io.Closeable5.0
■ void close() closes this Closeable. This method may throw an IOException.

java.io.Flushable5.0
■ void flush() flushes this Flushable.

java.lang.Readable5.0
■ int read(CharBuffer cb) attempts to read as many char values into cb as it can hold. Returns the number of values read, or -1 if no further values are available from this Readable.

java.lang.Appendable5.0
■ Appendable append(char c) ■ Appendable append(CharSequence cs) append the given code unit, or all code units in the given sequence, to this Appendable; return this.

java.lang.CharSequence1.4

- char charAt(int index)
  returns the code unit at the given index.
- int length()
  returns the number of code units in this sequence.
- CharSequence subSequence(int startIndex, int endIndex)
  returns a CharSequence consisting of the code units stored from index startIndex to endIndex - 1.
- String toString()
  returns a string consisting of the code units of this sequence.

2.1.3. Combining Input/Output Stream Filters

FileInputStream and FileOutputStream give you input and output streams attached to a disk file. You need to pass the file name or full path name of the file to the constructor. For example,

```
var fin = new FileInputStream("employee.dat");
```

looks for a file named employee.dat.

Relative path names start from the working directory. You can get this directory by a call to System.getProperty("user.dir"). It is set when the virtual machine starts and cannot be changed afterwards. If you start your program from the command line, the working directory is the current directory in the terminal. In an IDE, you can set the working directory in the run configuration.

Since the backslash character is the escape character in Java strings, be sure to use \\ for Windows-style path names (for example, C:\\Windows\\win.ini). In Windows, you can also use a single forward slash (C:/Windows/

win.ini) because most Windows file-handling system calls will interpret forward slashes as file separators. However, this is not recommended—the behavior of the Windows system functions is subject to change. Instead, for portable programs, use the file separator character for the platform on which your program runs. It is available as the constant string java.io.File.separator.

Like the abstract InputStream and OutputStream classes, these classes only support reading and writing at the byte level. That is, we can only read bytes and byte arrays from the object fin.

```
byte b = (byte) fin.read();
```

As you will see in the next section, if we just had a DataInputStream, we could read numeric types:

```
DataInputStream din = . . .;
double x = din.readDouble();
```

But just as the FileInputStream has no methods to read numeric types, the DataInputStream has no method to get data from a file.

Java uses a clever mechanism to separate two kinds of responsibilities. Some input streams (such as the FileInputStream and the input stream returned by the openStream method of the URL class) can retrieve bytes from files and other more exotic locations. Other input streams (such as the DataInputStream) can assemble bytes into more useful data types. The Java programmer has to combine the two. For example, to be able to read numbers from a file, first create a FileInputStream and then pass it to the constructor of a DataInputStream.

```
var fin = new FileInputStream("employee.dat");
var din = new DataInputStream(fin);
double x = din.readDouble();
```

If you look at #ch01fig01 again, you can see the classes FilterInputStream and FilterOutputStream. The subclasses of

these classes are used to add capabilities to input/output streams that process bytes.

You can add multiple capabilities by nesting the filters. For example, by default, input streams are not buffered. That is, every call to read asks the operating system to dole out yet another byte. It is more efficient to request blocks of data instead and store them in a buffer. If you want buffering and the data input methods for a file, use the following rather monstrous sequence of constructors:

```
var din = new DataInputStream(
   new BufferedInputStream(
      new FileInputStream("employee.dat")));
```

Notice that we put the DataInputStream last in the chain of constructors because we want to use the DataInputStream methods, and we want them to use the buffered read method.

Sometimes you'll need to keep track of the intermediate input streams when chaining them together. For example, when reading input, you often need to peek at the next byte to see if it is the value that you expect. Java provides the PushbackInputStream for this purpose.

```
var pbin = new PushbackInputStream(
   new BufferedInputStream(
      new FileInputStream("employee.dat")));
```

Now you can speculatively read the next byte

```
int b = pbin.read();
```

and throw it back if it isn't what you wanted.

```
if (b != '<') pbin.unread(b);
```

> The unread method doesn't actually alter the source from which the stream reads its bytes. Unread bytes are stored in a buffer and are returned by the next read operation. By default, the buffer has length 1, but you can

---

specify a longer buffer in the PushbackInputStream constructor.

Reading and unreading are the only methods that apply to a pushback input stream. If you want to look ahead and also read numbers, then you need both a pushback input stream and a data input stream reference.

```
var pbin = new PushbackInputStream(
   new BufferedInputStream(
      new FileInputStream("employee.dat")));
var din = new DataInputStream(pbin);
```

Of course, in the input/output libraries of other programming languages, niceties such as buffering and lookahead are automatically taken care of—so it is a bit of a hassle to resort, in Java, to combining stream filters. However, the ability to mix and match filter classes to construct useful sequences of input/output streams does give you an immense amount of flexibility. For example, you can read numbers from a compressed ZIP file by using the following sequence of input streams (see #ch01fig04):

```
var zin = new ZipInputStream(new FileInputStream("employee.zip")
var din = new DataInputStream(zin);
```

.

Figure 2.4: A sequence of filtered input streams

(See Section 2.2.3 for more on Java's handling of ZIP files.)

java.io.FileInputStream1.0

- FileInputStream(String name)
- FileInputStream(File file)
  create a new file input stream using the file whose path name is specified by the name string or the object of the legacy File class.

java.io.FileOutputStream1.0

- FileOutputStream(String name)
- FileOutputStream(String name, boolean append)
- FileOutputStream(File file)
- FileOutputStream(File file, boolean append)
  create a new file output stream specified by the name string or the file object. (The File class is described at the end of this chapter.) If the append parameter is true, an existing file with the same name will not be deleted and data will be added at the end of the file. Otherwise, this method deletes any existing file with the same name.

java.io.BufferedInputStream1.0

- BufferedInputStream(InputStream in)
  creates a buffered input stream. A buffered input stream reads bytes from a stream without causing a device access every time. When the buffer is empty, a new block of data is read into the buffer.

java.io.BufferedOutputStream1.0

- BufferedOutputStream(OutputStream out)
  creates a buffered output stream. A buffered output stream collects bytes to be written without causing a device access every time. When the buffer fills up or when the stream is flushed, the data are written.

java.io.PushbackInputStream1.0

- PushbackInputStream(InputStream in)
- PushbackInputStream(InputStream in, int size)
  construct an input stream with one-byte lookahead or a pushback buffer of specified size.

- void unread(int b)
  pushes back a byte, which is retrieved again by the next call to read.

2.1.4. Text Input and Output

When saving data, you have the choice between binary and text formats. For example, if the integer 1234 is saved in binary, it is written as the sequence of bytes 00 00 04 D2 (in hexadecimal notation). In text format, it is saved as the string "1234". Although binary I/O is fast and efficient, it is not easily readable by humans. We first discuss text I/O and cover binary I/O in Section 2.2.

When saving text strings, you need to consider the character encoding. In the UTF-16 encoding that Java uses internally, the string "José" is encoded as 00 4A 00 6F 00 73 00 E9 (in hex). However, many programs expect that text files use a different encoding. In UTF-8, the encoding most commonly used on the Internet, the string would be written as 4A 6F 73 C3 A9, without the zero bytes for the first three letters and with two bytes for the é character.

The OutputStreamWriter class turns an output stream of Unicode code units into a stream of bytes. Conversely, the InputStreamReader class turns an input stream that contains bytes (specifying characters in some character encoding) into a reader that emits Unicode code units.

Since Java 18, these classes use the UTF-8 encoding by default. Before Java 18, or if you need to support an archaic encoding, specify the encoding in the constructor, for example:

```
var in = new InputStreamReader(new FileInputStream("data.txt"), St
```

See Section 2.1.8 for more information on character encodings.

The Reader and Writer classes have only basic methods to read and write individual characters. As with streams, you use

subclasses for processing strings and numbers.

2.1.5. How to Read Text Input

You can read a short text file into a string like this:

```
String content = Files.readString(path);
```

But if you want the file as a sequence of lines, call

```
List<String> lines = Files.readAllLines(path);
```

If the file is large, process the lines lazily as a Stream<String>:

```
try (Stream<String> lines = Files.lines(path)) {
   . . .
}
```

If an IOException occurs as the stream fetches the lines, that exception is wrapped into an UncheckedIOException which is thrown out of the stream operation. This subterfuge is necessary because stream operations are not declared to throw any checked exceptions.

The easiest way to process arbitrary text is the Scanner class. This code processes all lines in a file:

```
try (var in = new Scanner(path)) {
   while (in.hasNextLine()) {
      String line = in.nextLine();
      // Process line
   }
}
```

You can also use a scanner to read tokens—strings that are separated by a delimiter. The default delimiter is whitespace. You can change the delimiter to any regular expression. For example, here is how to configure a scanner so that it uses any non-Unicode letters as delimiters.

```
Scanner in = . . .;
in.useDelimiter("\\PL+");
```

This scanner now accepts tokens consisting only of Unicode letters.

Calling the next method yields the next token:

```
while (in.hasNext()) {
   String word = in.next();
   . . .
}
```

Alternatively, you can obtain a stream of all tokens as

```
Stream<String> words = in.tokens();
```

> In legacy code, you may find code that reads lines by repeatedly calling the readLine method of a BufferedReader. Nowadays, there is no good reason to do that.

The nextInt and nextDouble methods yield the next integer or floating-point number in the input. Call hasNextInt or hasNextDouble first to make sure that the next token is a number.

> The next, nextInt, and nextDouble methods only consume leading whitespace and one token. Trailing whitespace is left in the input. Keep this in mind if you also call to nextLine. If your input is:
>
> ```
> 29.95
> Blackwell Toaster
> ```
>
> and you call
>
> ```
> double price = in.nextDouble();
> double description = in.nextLine(); // ERROR
> ```

then description will be empty! The call to nextLine only reads the newline following the number.

Simply add another call to nextLine to consume the trailing whitespace.

2.1.6. How to Write Text Output

For text output, use a PrintWriter. That class has methods to print strings and numbers in text format. In order to print to a file, construct a PrintWriter from a file name:

```
var out = new PrintWriter("employee.txt");
```

Curiously, you cannot construct a PrintWriter from a Path.

Before Java 18, be sure to specify the character encoding as a second argument:

```
var out = new PrintWriter("employee.txt", StandardCharsets.U
```

Otherwise, the platform encoding is used, making your program's behavior dependent on the configuration of the host operating system..

To write to a PrintWriter, use the print and println methods. You can use these methods to print numbers (int, short, long, float, double), characters, boolean values, strings, and objects.

For example, consider this code:

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

This writes the characters

Harry Hacker 75000.0

to the writer out. The characters are then converted to bytes and end up in the file employee.txt.

The println method adds the correct end-of-line character for the target system ("\r\n" on Windows, "\n" on UNIX) to the line. This is the string obtained by the call System.getProperty("line.separator").

If the writer is set to autoflush mode, all characters in the buffer are sent to their destination whenever println is called. (Print writers are always buffered.) By default, autoflushing is not enabled. You can enable or disable autoflushing by using the PrintWriter(Writer writer, boolean autoFlush) constructor:

```
var out = new PrintWriter(
   new OutputStreamWriter(
      new FileOutputStream("employee.txt")),
   true); // autoflush
```

The print methods don't throw exceptions. You can call the checkError method to see if something went wrong with the output stream.

Java veterans might wonder what happened to the PrintStream class and to System.out. In Java 1.0, the PrintStream class simply truncated all Unicode characters to ASCII characters by dropping the top byte. (At the time, Unicode was still a 16-bit encoding.) Clearly, that was not a clean or portable approach, and it was fixed with the introduction of readers and writers in Java 1.1. For compatibility with existing code, System.in, System.out, and System.err are still input/output streams, not readers and writers. But now the PrintStream class internally converts Unicode characters to the default character encoding in the same way the PrintWriter does. Objects of type PrintStream act exactly like print writers when you use the print and println methods, but unlike print writers they allow you to output raw bytes

---

with the write(int) and write(byte[]) methods.

java.io.PrintWriter1.1

- PrintWriter(OutputStream out)
- PrintWriter(OutputStream out, boolean autoFlush)
- PrintWriter(Writer writer)
- PrintWriter(Writer writer, boolean autoFlush)
  create a new PrintWriter that writes to the given output stream or writer. If autoFlush is specified as true, println and printf flush the output.
- PrintWriter(String filename, String encoding)
- PrintWriter(File file, String encoding)
  create a new PrintWriter that writes to the given file, using the given character encoding.
- void print(Object obj)
  prints an object by printing the string resulting from toString.
- void print(String s)
  prints a string containing Unicode code units.
- void println(String s)
  prints a string followed by a line terminator. Flushes the output stream if it is in autoflush mode.
- void print(char[] s)
  prints all Unicode code units in the given array.
- void print(char c)
  prints a Unicode code unit.
- void print(int i)
- void print(long l)
- void print(float f)
- void print(double d)
- void print(boolean b)
  print the given value in text format.
- void printf(String format, Object... args)
  prints the given values as specified by the format string. See Chapter 3 of Volume I for the specification of the format string.
- boolean checkError()
  returns true if a formatting or output error occurred. Once

the output stream has encountered an error, it is tainted and all calls to checkError return true.

2.1.7. Saving Objects in Text Format

In this section, we discuss an example program that stores an array of Employee records in a text file. Each record is stored in a separate line. Instance fields are separated from each other by delimiters. A vertical bar (|) is used as a delimiter. (A colon (:) is another popular choice. Part of the fun is that everyone uses a different delimiter.) Naturally, we punt on the issue of what might happen if a | actually occurs in one of the strings we save.

Here is a sample set of records:

```
Harry Hacker|35500.0|1989-10-01
Carl Cracker|75000.0|1987-12-15
Tony Tester|38000.0|1990-03-15
```

Writing records is simple. Since we write to a text file, we use the PrintWriter class. We simply write all fields, followed by either a | or, for the last field, a newline character. This work is done in the following method:

```
public static void writeEmployee(PrintWriter out, Employee e) {
   out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay())
}
```

To read records, we read in a line at a time and separate the fields. We use a scanner to read each line and then split the line into tokens with the String.split method.

```
public static Employee readEmployee(Scanner in) {
   String line = in.nextLine();
   String[] tokens = line.split("\\|");
   String name = tokens[0];
   double salary = Double.parseDouble(tokens[1]);
   LocalDate hireDate = LocalDate.parse(tokens[2]);
```

```
    int year = hireDate.getYear();
    int month = hireDate.getMonthValue();
    int day = hireDate.getDayOfMonth();
    return new Employee(name, salary, year, month, day);
}
```

The parameter of the split method is a regular expression describing the separator. We discuss regular expressions in more detail at the end of this chapter. As it happens, the vertical bar character has a special meaning in regular expressions, so it needs to be escaped with a \ character. That character needs to be escaped by another \, yielding the "\\|" expression.

The complete program is in Listing 2.1. The static method

```
void writeData(Employee[] e, PrintWriter out)
```

first writes the length of the array, then writes each record. The static method

```
Employee[] readData(Scanner in)
```

first reads in the length of the array, then reads in each record. This turns out to be a bit tricky:

```
int n = in.nextInt();
in.nextLine(); // consume newline
var employees = new Employee[n];
for (int i = 0; i < n; i++) {
   employees[i] = readEmployee(in);
}
```

The call to nextInt reads the array length but not the trailing newline character. We must consume the newline so that the readEmployee method can get the next input line when it calls the nextLine method.

Listing 2.1 **v2ch02/textFile/TextFileDemo.java**

```
package v2ch02.textFile;
```

```
import com.horstmann.corejava.*;
import module java.base;

class TextFileDemo {
   void main() throws Exception {
      var staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      // save all employee records to the file employee.dat
      try (var out = new PrintWriter("employee.dat")) {
         writeData(staff, out);
      }

      // retrieve all records into a new array
      try (var in = new Scanner(new FileInputStream("employee.dat"))) {
         Employee[] newStaff = readData(in);

         // print the newly read employee records
         for (Employee e : newStaff)
            IO.println(e);
      }
   }

   void writeData(Employee[] employees, PrintWriter out) throws IOException
      // write number of employees
      out.println(employees.length);

      for (Employee e : employees)
         writeEmployee(out, e);
   }

   Employee[] readData(Scanner in) {
      // retrieve the array size
      int n = in.nextInt();
      in.nextLine(); // consume newline

      var employees = new Employee[n];
      for (int i = 0; i < n; i++) {
         employees[i] = readEmployee(in);
      }
```

```
      return employees;
   }

   void writeEmployee(PrintWriter out, Employee e) {
      out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
   }

   Employee readEmployee(Scanner in) {
      String line = in.nextLine();
      String[] tokens = line.split("\\|");
      String name = tokens[0];
      double salary = Double.parseDouble(tokens[1]);
      LocalDate hireDate = LocalDate.parse(tokens[2]);
      int year = hireDate.getYear();
      int month = hireDate.getMonthValue();
      int day = hireDate.getDayOfMonth();
      return new Employee(name, salary, year, month, day);
   }
}
```

2.1.8. Character Encodings

Each Unicode "code point" has a 21-bit integer number. There are different character encodings—methods for packaging those 21-bit numbers into bytes.

The most common encoding is UTF-8, which encodes each Unicode code point into a sequence of one to four bytes. Table 2.1 shows in detail how to encode a 21-bit code point with bits $a_{20}a_{19}\ldots a_1a_0$. UTF-8 has the advantage that the characters of the traditional ASCII character set, which contains all characters used in English, only take up one byte each.

Character Range	Encoding
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

Table 2.1: UTF-8 Encoding

Encoding	Range
Two bytes	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
Three bytes	110110$b_{19}b_{18}$ $b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}$ 110111a_9a_8 $a_7a_6a_5a_4a_3a_2a_1a_0$ where $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

Table 2.2: UTF-16 Encoding

Another common encoding is UTF-16, which encodes each Unicode code point into one or two 16-bit values (see Table 2.2). This is the encoding used in Java strings. Actually, there are two forms of UTF-16, called “big-endian” and “little-endian.” Consider the 16-bit value 0x2122. In the big-endian format, the more significant byte comes first: 0x21 followed by 0x22. In the little-endian format, it is the other way around: 0x22 0x21. To indicate which of the two is used, a file can start with the “byte order mark” or BOM, the 16-bit quantity 0xFEFF. A reader can use this value to determine the byte order and then discard it.

> Some programs add a BOM at the beginning of UTF-8 encoded files. Clearly, this is unnecessary since there are no byte ordering issues in UTF-8. But the Unicode standard allows it, and some programmers consider it a good idea since it leaves little doubt about the encoding. Any BOM is supposed to be ignored when reading a UTF-8 encoded file. Sadly, Java does not do that, and bug reports against this issue are closed as “will not fix.” Your best bet is to strip out any leading \uFEFF that you find in your input.

In addition to the UTF encodings, there are partial encodings that cover a character range suitable for a given user population. For example, ISO 8859-1 is a one-byte code that includes accented characters used in Western European languages. Shift-JIS is a variable-length code for Japanese characters. A number of these encodings are still in use.

There is no reliable way to automatically detect the character encoding from a stream of bytes. Sometimes, the encoding is

specified elsewhere. For example, when reading a web page, check the Content-Type header.

The StandardCharsets class has static variables of type Charset for the character encodings that every Java virtual machine must support:

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.UTF_32
StandardCharsets.UTF_32BE
StandardCharsets.UTF_32LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

To obtain the Charset for another encoding, use the static forName method:

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

The static method Charset.availableCharsets returns all available Charset instances, as a map from canonical names to Charset objects.

Use a Charset object whenever you need to specify a character encoding. For example, you can turn an array of bytes into a string as

```
var str = new String(bytes, StandardCharsets.ISO_8859_1);
```

Many methods allow you to specify a character encoding as a string instead of a CharSet. For the standard character encodings, use the StandardCharsets constants, so that any spelling errors are caught at compile time.

Up to Java 17, calling Charset.forName with the name "default" yielded StandardCharsets.US_ASCII! Nowadays,

---

the call results in an UnsupportedCharsetException.

2.1.9. Default Character Encodings

If you do not specify a character encoding, and an operation needs to convert between bytes and strings, it will pick a default. For many methods in the java.nio.file package, that default is UTF-8.

But other methods use the default charset when no character encoding is specified. These include:

- The methods of the java.io reader and writer classes
- The String(byte[]) constructor and the getBytes method of the String class
- Scanner and Formatter in the java.util package
- URLEncoder and URLDecoder in the java.net package

Prior to Java 18, the default charset was the native encoding, determined from the system executing the Java virtual machine. On Linux and Mac OS, that is UTF-8, unless it was specifically configured otherwise. However, on Windows, the native encoding is in most cases an archaic encoding such as Cp1252. (As of Java 17, the system property native.encoding yields the name of the native encoding.)

Prior to Java 18, the Oracle implementation of Java had a system property file.encoding for overriding the default charset. This was not an officially supported property, but it was nevertheless widely used.

As of Java 18, this property has become standardized. You can set the file.encoding property on the command line to these values:

- COMPAT: Sets the default charset to the native encoding, and the file.encoding system property to its name.
- UTF-8: Sets the default charset to UTF-8.
- Another character encoding such as US-ASCII: For some unspecified legacy charset names, it may set the default charset to the given one.

The file.encoding property must be set on the command line. Setting it via System.setProperty does not update the default charset.

The actually used default charset is returned by the static method Charset.defaultCharset.

Finally, the Windows console uses an even more archaic character encoding than the one used for file encoding. You may want to override that when redirecting output to a file. As of Java 18, you can do so by setting the system properties stdout.encoding and stderr.encoding.

The naming of these system properties is perhaps confusing. The encoding does not apply to stdout and stderr (which are byte streams) but to the System.out and System.err objects. Both are instances of PrintStream, whose print and println methods need an encoding that turns characters to bytes.

Java 25 adds the stdin.encoding property, specifying the encoding when reading from System.in. It is used by the IO and Console classes, but not by Scanner.

To quickly see your current settings for the various encodings, run:

```
java -XshowSettings:properties
```

Look for the properties whose names end in .encoding. Here are typical property values for Java 25:

Property	Windows	Linux/Mac OS
native.encoding	Cp1252	UTF-8
file.encoding	UTF-8	UTF-8
stdin.encoding	cp437	UTF-8
stdout.encoding	cp437	UTF-8
stderr.encoding	cp437	UTF-8

2.1.10. Reading Character Input

If you read a file with a structured format such as JSON or XML, you will use a parser that someone wrote who understands the fiddly details of that format. Such a parser typically reads a character at a time.

In the uncommon case that you need to implement such a parser, use a BufferedReader for efficiency. Keep calling its read method, which yields a char value or -1 at the end of input. The reader yields UTF-16 code units. If you are interested in code points, you need to handle the UTF-16 encoding:

```
int ch = reader.read();
if (ch != -1) {
   int codePoint;
   if (Character.isHighSurrogate((char) ch)) {
      int ch2 = reader.read();
      if (Character.isLowSurrogate((char) ch2))
         codePoint = Character.toCodePoint(ch, ch2);
      else
         throw new MalformedInputException(2);
   }
   else
      codePoint = ch;
}
```

The Character class contains methods to tell whether a particular code point has a given property. For example,

```
Character.isLetter(codePoint)
```

returns true if codePoint is a letter in some language. Here are some other classification methods:

```
isUpperCase
isLowerCase
isDigit
isSpaceChar
isEmoji
```

These methods use the rules of the Unicode standard. Others refer to the rules of the Java language:

isJavaIdentifierStart
isJavaIdentifierPart
isWhitespace

After analyzing the code points, you often need to store them in strings, converting them back to UTF-16. The appendCodePoint method of the StringBuilder class turns a code point into one or two char values which are appended to the builder.

2.2. Reading and Writing Binary Data

Text format is convenient for testing and debugging because it is humanly readable, but it is not as efficient as transmitting data in binary format. In the following sections, you will learn how to perform input and output with binary data.

2.2.1. The DataInput and DataOutput Interfaces

The DataOutput interface defines the following methods for writing a number, a character, a boolean value, or a string in binary format:

writeChars writeFloat
writeByte writeDouble
writeInt writeChar
writeShort writeBoolean
writeLong writeUTF

For example, writeInt always writes an integer as a 4-byte binary quantity regardless of the number of digits, and writeDouble always writes a double as an 8-byte binary quantity. The resulting output is not human-readable, but it will use the same space for each value of a given type and reading it back in will be faster than parsing text.

There are two different methods of storing integers and

floating-point numbers in memory, depending on the processor you are using. Suppose, for example, you are working with a 4-byte int, such as the decimal number 1234, or 4D2 in hexadecimal (1234 = 4 × 256 + 13 × 16 + 2). This value can be stored in such a way that the first of the four bytes in memory holds the most significant byte (MSB) of the value: 00 00 04 D2. This is the so-called big-endian method. Or, we can start with the least significant byte (LSB) first: D2 04 00 00. This is called, naturally enough, the little-endian method. For example, the SPARC uses big-endian; Intel processors, little-endian. This can lead to problems. When a file is saved from a C or C++ file, the data are saved exactly as the processor stores them. That makes it challenging to move even the simplest data files from one platform to another. In Java, all values are written in the big-endian fashion, regardless of the processor. That makes Java data files platform-independent.

The writeUTF method writes string data using a modified version of the 8-bit Unicode Transformation Format. Instead of simply using the standard UTF-8 encoding, sequences of Unicode code units are first represented in UTF-16, and then the result is encoded using the UTF-8 rules. This modified encoding is different for characters with codes higher than 0xFFFF. It is used for backward compatibility with virtual machines that were built when Unicode had not yet grown beyond 16 bits.

Since nobody else uses this modification of UTF-8, you should only use the writeUTF method to write strings intended for a Java virtual machine—for example, in a program that generates bytecodes. Use the writeChars method for other purposes.

To read the data back in, use the following methods defined in the DataInput interface:

readInt	readDouble
readShort	readChar
readLong	readBoolean

readFloat readUTF

The DataInputStream class implements the DataInput interface. To read binary data from a file, combine a DataInputStream with a source of bytes such as a FileInputStream:

```
var in = new DataInputStream(new FileInputStream("employee.dat"
```

Similarly, to write binary data, use the DataOutputStream class that implements the DataOutput interface:

```
var out = new DataOutputStream(new FileOutputStream("employee.
```

java.io.DataInput1.0

- boolean readBoolean()
- byte readByte()
- char readChar()
- double readDouble()
- float readFloat()
- int readInt()
- long readLong()
- short readShort()
  read in a value of the given type.
- void readFully(byte[] b)
  reads bytes into the array b, blocking until all bytes are read.
- void readFully(byte[] b, int off, int len)
  places up to len bytes into the array b, starting at off, blocking until all bytes are read.
- String readUTF()
  reads a string of characters in the “modified UTF-8” format.
- int skipBytes(int n)
  skips n bytes, blocking until all bytes are skipped.

java.io.DataOutput1.0

- void writeBoolean(boolean b)
- void writeByte(int b)
- void writeChar(int c)
- void writeDouble(double d)
- void writeFloat(float f)
- void writeInt(int i)
- void writeLong(long l)
- void writeShort(int s)
  write a value of the given type.
- void writeChars(String s)
  writes all characters in the string.
- void writeUTF(String s)
  writes a string of characters in the “modified UTF-8” format.

2.2.2. Random-Access Files

The RandomAccessFile class lets you read or write data anywhere in a file. Disk files are random-access, but input/output streams that communicate with a network socket are not. You can open a random-access file either for reading only or for both reading and writing; specify the option by using the string "r" (for read access) or "rw" (for read/write access) as the second argument in the constructor.

```
var in = new RandomAccessFile("employee.dat", "r");
var inOut = new RandomAccessFile("employee.dat", "rw");
```

There are also modes "rwd" and "rws" that flush updates of the data, or the data and metadata, to disk. This is slower but can prevent data loss when the program or computer crashes.

When you open an existing file as a RandomAccessFile, it does not get deleted.

A random-access file has a file pointer that indicates the position of the next byte to be read or written. The seek method can be used to set the file pointer to an arbitrary byte position within the file, specified as a long value.

The getFilePointer method returns the current position of the file pointer.

The RandomAccessFile class implements both the DataInput and DataOutput interfaces. To read and write from a random-access file, use methods such as readInt/writeInt and readChar/writeChar that we discussed in the preceding section.

Let's walk through an example program that stores employee records in a random-access file. Each record will have the same size. This makes it easy to read an arbitrary record. Suppose you want to position the file pointer to the third record. Simply set the file pointer to the appropriate byte position and start reading.

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
var e = new Employee();
e.readData(in);
```

If you want to modify the record and save it back into the same location, remember to set the file pointer back to the beginning of the record:

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

To determine the total number of bytes in a file, use the length method. The total number of records is the length divided by the size of each record.

```
long nbytes = in.length(); // length in bytes
int nrecords = (int) (nbytes / RECORD_SIZE);
```

Integers and floating-point values have a fixed size in binary format, but we have to work harder for strings. We provide two helper methods to write and read strings of a fixed size.

The writeFixedString writes the specified number of code units, starting at the beginning of the string. If there are too few code units, the method pads the string, using zero values.

```
public static void writeFixedString(String s, int size, DataOutput out)
      throws IOException {
   for (int i = 0; i < size; i++) {
      char ch = 0;
      if (i < s.length()) ch = s.charAt(i);
      out.writeChar(ch);
   }
}
```

The readFixedString method reads characters from the input stream until it has consumed size code units or until it encounters a character with a zero value. Then, it skips past the remaining zero values in the input field. For added efficiency, this method uses the StringBuilder class to read in a string.

```
public static String readFixedString(int size, DataInput in)
      throws IOException {
   var b = new StringBuilder(size);
   int i = 0;
   var done = false;
   while (!done && i < size) {
      char ch = in.readChar();
      i++;
      if (ch == 0) done = true;
      else b.append(ch);
   }
   in.skipBytes(2 * (size - i));
   return b.toString();
}
```

We placed the writeFixedString and readFixedString methods inside the DataIO helper class.

To write a fixed-size record, we simply write all fields in binary.

```
DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
LocalDate hireDay = e.getHireDay();
out.writeInt(hireDay.getYear());
out.writeInt(hireDay.getMonthValue());
out.writeInt(hireDay.getDayOfMonth());
```

Reading the data back is just as simple.

```
String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
double salary = in.readDouble();
int y = in.readInt();
int m = in.readInt();
int d = in.readInt();
```

Let us compute the size of each record. We will use 40 characters for the name strings. Therefore, each record will contain 100 bytes:

- 40 characters = 80 bytes for the name
- 1 double = 8 bytes for the salary
- 3 int = 12 bytes for the date

The program shown in Listing 2.2 writes three records into a data file and then reads them from the file in reverse order. To do this efficiently requires random access—we need to get to the last record first.

Listing 2.2 **v2ch02/randomAccess/RandomAccessDemo.java**

```
package v2ch02.randomAccess;

import module java.base;
import com.horstmann.corejava.*;

public class RandomAccessDemo {
   final int NAME_SIZE = 40;
   final int RECORD_SIZE = 2 * NAME_SIZE + 8 + 4 + 4 + 4;

   void main() throws Exception {
      var staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      try (var out = new DataOutputStream(new FileOutputStream("employee.d
         // save all employee records to the file employee.dat
```

```
      for (Employee e : staff)
         writeData(out, e);
   }

   try (var in = new RandomAccessFile("employee.dat", "r")) {
      // retrieve all records into a new array

      // compute the array size
      int n = (int) (in.length() / RECORD_SIZE);
      var newStaff = new Employee[n];

      // read employees in reverse order
      for (int i = 0; i < n; i++) {
         in.seek((n - 1 - i) * RECORD_SIZE);
         newStaff[i] = readData(in);
      }

      // print the newly read employee records
      for (Employee e : newStaff)
         IO.println(e);
   }
}

/**
 * Writes employee data to a data output.
 * @param out the data output
 * @param e the employee
 */
void writeData(DataOutput out, Employee e) throws IOException {
   DataIO.writeFixedString(e.getName(), NAME_SIZE, out);
   out.writeDouble(e.getSalary());

   LocalDate hireDay = e.getHireDay();
   out.writeInt(hireDay.getYear());
   out.writeInt(hireDay.getMonthValue());
   out.writeInt(hireDay.getDayOfMonth());
}

/**
 * Reads employee data from a data input.
 * @param in the data input
 * @return the employee
 */
Employee readData(DataInput in) throws IOException {
```

```
      String name = DataIO.readFixedString(NAME_SIZE, in);
      double salary = in.readDouble();
      int y = in.readInt();
      int m = in.readInt();
      int d = in.readInt();
      return new Employee(name, salary, y, m, d);
   }
}
```

java.io.RandomAccessFile1.0

- RandomAccessFile(String file, String mode)
- RandomAccessFile(File file, String mode)
  open the given file for random access. The mode string is "r" for read-only mode, "rw" for read/write mode, "rws" for read/write mode with synchronous disk writes of data and metadata for every update, and "rwd" for read/write mode with synchronous disk writes of data only.
- long getFilePointer()
  returns the current location of the file pointer.
- void seek(long pos)
  sets the file pointer to pos bytes from the beginning of the file.
- long length()
  returns the length of the file in bytes.

2.2.3. ZIP Archives

ZIP archives store one or more files in a (usually) compressed format. Each ZIP archive has a header with information such as the name of each file and the compression method that was used. In Java, you can use a ZipInputStream to read a ZIP archive. You need to look at the individual entries in the archive. The getNextEntry method returns an object of type ZipEntry that describes the entry. Read from the stream until the end, which is actually the end of the current entry. Then call closeEntry to read the next entry. Do not close the ZipInputStream until you read the last entry. Here is a typical code sequence to read through a ZIP file:

```
var zin = new ZipInputStream(new FileInputStream(zipname));
boolean done = false;
while (!done) {
   ZipEntry entry = zin.getNextEntry();
   if (entry == null) done = true;
   else {
      read the contents of zin
      zin.closeEntry();
   }
}
zin.close();
```

To write a ZIP file, use a ZipOutputStream. For each entry that you want to place into the ZIP file, create a ZipEntry object. Pass the file name to the ZipEntry constructor; it sets the other parameters such as file date and decompression method. You can override these settings if you like. Then, call the putNextEntry method of the ZipOutputStream to begin writing a new file. Send the file data to the ZIP output stream. When done, call closeEntry. Repeat for all the files you want to store. Here is a code skeleton:

```
var fout = new FileOutputStream("test.zip");
var zout = new ZipOutputStream(fout);
for all files {
   var ze = new ZipEntry(filename);
   zout.putNextEntry(ze);
   send data to zout
   zout.closeEntry();
}
zout.close();
```

JAR files (which were discussed in Chapter 4 of Volume I) are simply ZIP files with a special entry—the so-called manifest. Use the JarInputStream and JarOutputStream classes to read and write the manifest entry.

ZIP input streams are an example of the power of the stream abstraction. When you read data stored in compressed form,

you don't need to worry that the data are being decompressed as they are being requested. Moreover, the source of the bytes in a ZIP stream need not be a file—the ZIP data can come from a network connection.

> Section 2.3.9 shows how to access a ZIP archive without a special API, using the FileSystem class.

---

java.util.zip.ZipInputStream1.1

- ZipInputStream(InputStream in)
  creates a ZipInputStream that allows you to inflate data from the given InputStream.
- ZipEntry getNextEntry()
  returns a ZipEntry object for the next entry, or null if there are no more entries.
- void closeEntry()
  closes the current open entry in the ZIP file. You can then read the next entry by using getNextEntry().

java.util.zip.ZipOutputStream1.1

- ZipOutputStream(OutputStream out)
  creates a ZipOutputStream that you can use to write compressed data to the specified OutputStream.
- void putNextEntry(ZipEntry ze)
  writes the information in the given ZipEntry to the output stream and positions the stream for the data. The data can then be written by calling the write() method.
- void closeEntry()
  closes the currently open entry in the ZIP file. Use the putNextEntry method to start the next entry.
- void setLevel(int level)
  sets the default compression level of subsequent DEFLATED entries to a value from Deflater.NO_COMPRESSION to Deflater.BEST_COMPRESSION. The default value is

Deflater.DEFAULT_COMPRESSION. Throws an IllegalArgumentException if the level is not valid.
- void setMethod(int method)
  sets the default compression method for this ZipOutputStream for any entries that do not specify a method; can be either DEFLATED or STORED.

java.util.zip.ZipEntry1.1

- ZipEntry(String name)
  constructs a ZIP entry with a given name.
- long getCrc()
  returns the CRC32 checksum value for this ZipEntry.
- String getName()
  returns the name of this entry.
- long getSize()
  returns the uncompressed size of this entry, or -1 if the uncompressed size is not known.
- boolean isDirectory()
  returns true if this entry is a directory.
- void setMethod(int method)
  sets the compression method for the entry to DEFLATED or STORED.
- void setSize(long size)
  sets the size of this entry. Only required if the compression method is STORED.
- void setCrc(long crc)
  sets the CRC32 checksum of this entry. Use the CRC32 class to compute this checksum. Only required if the compression method is STORED.

2.3. Working with Files

You have learned how to read and write data from a file. However, there is more to file management than reading and writing. The Path interface and Files class encapsulate the functionality required to work with the file system on the

user's machine. For example, the Files class can be used to remove or rename a file, or to find out when a file was last modified. In other words, the input/output stream classes are concerned with the contents of files, whereas the classes that we discuss here are concerned with the storage of files on a disk.

The Path interface and Files class are much more convenient to use than the File class which dates back all the way to JDK 1.0. You should not use the File class except to interface with legacy code.

2.3.1. Paths

A Path is a sequence of directory names, optionally followed by a file name. The first component of a path may be a root component such as / or C:\. The permissible root components depend on the file system. A path that starts with a root component is absolute. Otherwise, it is relative. For example, here we construct an absolute and a relative path. For the absolute path, we assume a UNIX-like file system.

```
Path absolute = Path.of("/home", "harry");
Path relative = Path.of("myprog", "conf", "user.properties");
```

The static Path.of method receives one or more strings, which it joins with the path separator of the default file system (/ for a UNIX-like file system, \ for Windows). It then parses the result, throwing an InvalidPathException if the result is not a valid path in the given file system. The result is a Path object.

The of method can get a single string containing multiple components. For example, you can read a path from a configuration file like this:

```
String baseDir = props.getProperty("base.dir");
   // May be a string such as /opt/myprog or c:\Program Files\myprog
Path basePath = Path.of(baseDir); // OK that baseDir has separators
```

A path does not have to correspond to a file that actually

exists. It is merely an abstract sequence of names. As you will see in the next section, when you want to create a file, you first make a path and then call a method to create the corresponding file.

It is very common to combine or resolve paths. The call p.resolve(q) returns a path according to these rules:

- If q is absolute, then the result is q.
- if q does not have a root, then the result is obtained by joining p and q.
- Otherwise, the result depends on the rules of the file system.

For example, suppose your application needs to find its working directory relative to a given base directory that is read from a configuration file, as in the preceding example.

```
Path workRelative = Path.of("work");
Path workPath = basePath.resolve(workRelative);
```

There is a shortcut for the resolve method that takes a string instead of a path:

```
Path workPath = basePath.resolve("work");
```

The resolve method does not simplify the result. For example,

```
Path.of("/usr/bin").resolve("../local")
```

is a path /usr/bin/../local.

There is a convenience method resolveSibling that resolves against a path's parent, yielding a sibling path. For example, if workPath is /opt/myapp/work, the call

```
Path tempPath = workPath.resolveSibling("temp");
```

creates /opt/myapp/temp.

The opposite of resolve is relativize. The call p.relativize(r) yields the path q which, when resolved with p, yields r. For example, relativizing /home/harry against /home/fred/input.txt yields ../fred/input.txt. Here, we assume that .. denotes the parent directory in the file system.

The normalize method removes any redundant . and .. components (or whatever the file system may deem redundant). For example, normalizing the path /home/harry/../fred/./input.txt yields /home/fred/input.txt.

The toAbsolutePath method yields the absolute path of a given path, starting at a root component, such as /home/fred/input.txt or c:\Users\fred\input.txt.

The Path interface has many useful methods for taking paths apart. This code sample shows some of the most useful ones:

```
Path p = Path.of("/home", "fred", "myprog.properties");
Path parent = p.getParent(); // the path /home/fred
Path file = p.getFileName(); // the path myprog.properties
Path root = p.getRoot(); // the path /
```

The getFileName method simply returns the last component of the given path. If the path describes a directory, that's a directory name, not a file name.

As you have already seen in Volume I, you can construct a Scanner from a Path object:

```
var in = new Scanner(Path.of("/home/fred/input.txt"));
```

Occasionally, you may need to interoperate with legacy APIs that use the File class instead of the Path interface. The Path interface has a toFile method, and the File class has a toPath method.

---

java.nio.file.Path7

- static Path of(String first, String... more)11
  makes a path by joining the given strings.
- Path resolve(Path other)
- Path resolve(String other)
  If other is absolute, return other; otherwise, return the path obtained by joining this and other.
- Path resolve(Path first, Path... more)22
- Path resolve(String first, String... more)22
  resolve this path against first, then against the other paths.
- Path resolveSibling(Path other)
- Path resolveSibling(String other)
  If other is absolute, return other; otherwise, return the path obtained by joining the parent of this and other.
- Path relativize(Path other)
  returns the relative path that, when resolved with this, yields other.
- Path normalize()
  removes redundant path elements such as . and ..
- Path toAbsolutePath()
  returns this path if it is absolute; otherwise, this path resolved against the working directory.
- Path getParent()
  returns the parent, or null if this path has no parent.
- Path getFileName()
  returns the last component of this path, or null if this path has no components.
- Path getRoot()
  returns the root component of this path, or null if this path has no root components.
- toFile()
  makes a File from this path.

java.io.File1.0

- Path toPath()7
  makes a Path from this file.

2.3.2. Reading and Writing Files

The Files class makes quick work of common file operations. For example, you can easily read the entire contents of a file as a byte array, string, list of lines, or stream of lines:

```
byte[] bytes = Files.readAllBytes(path);
String content = Files.readString(path);
List<String> lines = Files.readAllLines(path);
Stream<String> lineStream = Files.lines(path);
```

Conversely, for writing:

```
Files.write(path, bytes);
Files.writeString(path, content);
Files.write(path, lines);
```

To append to a given file, use

```
Files.write(path, content, StandardOpenOption.APPEND);
```

The call

```
long pos = Files.mismatch(path1, path2);
```

yields the position of the first byte where the file contents differ.

To get the MIME type of a file (such as "text/html" or "image/png"), call

```
String mimeType = Files.probeContentType(path);
```

The exact behavior depends on the Java implementation. Each implementation has a collection of FileTypeDetector instances that are tried in unspecified order. Each detector may examine the file extension, OS-specific metadata, or the first few bytes of the contents.

The following methods allow you to interact with an API that uses the classic input/output streams or readers/writers:

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path);
Writer out = Files.newBufferedWriter(path);
```

These convenience methods save you from dealing with FileInputStream, FileOutputStream, BufferedReader, or BufferedWriter.

java.nio.file.Files[7]

- static byte[] readAllBytes(Path path)
- static String readString(Path path, Charset charset)
- static List<String> readAllLines(Path path, Charset charset)
  read the contents of a file.
- static Path write(Path path, byte[] contents, OpenOption... options)
- static Path write(Path path, String contents, Charset charset, OpenOption... options)
- static Path writeString(Path path, CharSequence contents, Charset cs, OpenOption... options)[11]
- static Path write(Path path, Iterable<? extends CharSequence> contents, Charset cs, OpenOption options)
  write the given contents to a file and return path.
- static long mismatch(Path path, Path path2)[12]
  returns the position of the first byte where the file contents differ.
- static String probeContentType(Path path)[12]
  returns the best guess of the installed file type detectors for the MIME type of the file contents, or null if the type could not be determined.
- static InputStream newInputStream(Path path, OpenOption... options)
- static OutputStream newOutputStream(Path path, OpenOption... options)
- static BufferedReader newBufferedReader(Path path, Charset charset)
- static BufferedWriter newBufferedWriter(Path path,

Charset charset, OpenOption... options)
open a file for reading or writing.

2.3.3. Creating Files and Directories

To create a new directory, call

```
Files.createDirectory(path);
```

All but the last component in the path must already exist. To create intermediate directories as well, use

```
Files.createDirectories(path);
```

You can create an empty file with

```
Files.createFile(path);
```

The call throws an exception if the file already exists. The check for existence and creation are atomic. If the file doesn't exist, it is created before anyone else has a chance to do the same.

There are convenience methods for creating a temporary file or directory in a given or system-specific location.

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = Files.createTempDirectory(prefix);
```

Here, dir is a Path, and prefix/suffix are strings which may be null. For example, the call Files.createTempFile(null, ".txt") might return a path such as /tmp/1234405522364837194.txt.

When you create a file or directory, you can specify attributes, such as owners or permissions. However, the details depend on the file system, and we won't cover them here.

java.nio.file.Files[7]

- static Path createFile(Path path, FileAttribute<?>... attrs)
- static Path createDirectory(Path path, FileAttribute<?>... attrs)
- static Path createDirectories(Path path, FileAttribute<?>... attrs)
  create a file or directory. The createDirectories method creates any intermediate directories as well.
- static Path createLink(Path link, Path existing)
- static Path createSymbolicLink(Path link, Path target, FileAttribute<?>... attrs)
  creates a “hard” or symbolic link to the given target.
- static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)
- static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>... attrs)
- static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)
- static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)
  create a temporary file or directory, in a location suitable for temporary files or in the given parent directory. Return the path to the created file or directory.

2.3.4. Copying, Moving, and Deleting Files

To copy a file from one location to another, simply call

```
Files.copy(fromPath, toPath);
```

To move the file instead, call

```
Files.move(fromPath, toPath);
```

The copy or move will fail if the target exists. If you want to overwrite an existing target, use the REPLACE_EXISTING option. If you want to copy all file attributes, use the COPY_ATTRIBUTES option. You can supply both like this:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTIN
```

```
StandardCopyOption.COPY_ATTRIBUTES);
```

You can specify that a move should be atomic. Then you are assured that either the move completed successfully, or the source continues to be present. Use the ATOMIC_MOVE option:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

You can also copy an input stream to a Path, which just means saving the input stream to disk. Similarly, you can copy a Path to an output stream. Use the following calls:

```
Files.copy(inputStream, toPath);
Files.copy(fromPath, outputStream);
```

As with the other calls to copy, you can supply copy options as needed.

Finally, to delete a file, simply call

```
Files.delete(path);
```

This method throws an exception if the file doesn't exist, so instead you may want to use

```
boolean deleted = Files.deleteIfExists(path);
```

The deletion methods can also be used to remove an empty directory.

See Table 2.4 for a summary of the options that are available for file operations.

Option	Description
~~StandardOpenOption; use~~ with newBufferedWriter, newInputStream, ~~newOutputStream, write.~~	
~~READ~~	~~Open for reading.~~
~~WRITE~~	~~Open for writing.~~
~~APPEND~~	~~If opened for writing,~~ append to the end of the

	file.
TRUNCATE_EXISTING	If opened for writing, remove existing contents.
CREATE_NEW	Create a new file and fail if it exists.
CREATE	Atomically create a new file if it doesn't exist.
DELETE_ON_CLOSE	Make a “best effort” to delete the file when it is closed.
SPARSE	A hint to the file system that this file will be sparse.
DSYNC or SYNC	Requires that each update to the file data or data and metadata be written synchronously to the storage device.
StandardCopyOption; use with copy, move.	
ATOMIC_MOVE	Move the file atomically.
COPY_ATTRIBUTES	Copy the file attributes.
REPLACE_EXISTING	Replace the target if it exists.
LinkOption; use with all of the above methods and exists, isDirectory, isRegularFile.	
NOFOLLOW_LINKS	Do not follow symbolic links.
FileVisitOption; use with find, walk, walkFileTree	
FOLLOW_LINKS	Follow symbolic links.

Table 2.4: Standard Options for File Operations

java.nio.file.Files[7]

- static Path copy(Path from, Path to, CopyOption... options)
- static Path move(Path from, Path to, CopyOption... options)

copy or move from to the given target location and return

to.
- static long copy(InputStream from, Path to, CopyOption... options)
- static long copy(Path from, OutputStream to, CopyOption... options)
  copy from an input stream to a file, or from a file to an output stream, returning the number of bytes copied.
- static void delete(Path path)
- static boolean deleteIfExists(Path path)
  delete the given file or empty directory. The first method throws an exception if the file or directory doesn't exist. The second method returns false in that case.

2.3.5. Getting File Information

The following static methods return a boolean value to check a property of a path:

- exists
- isHidden
- isReadable, isWritable, isExecutable
- isRegularFile, isDirectory, isSymbolicLink

The size method returns the number of bytes in a file.

```
long fileSize = Files.size(path);
```

The getOwner method returns the owner of the file, as an instance of java.nio.file.attribute.UserPrincipal.

All file systems report a set of basic attributes, encapsulated by the BasicFileAttributes interface, which partially overlaps with that information. The basic file attributes are

- The times at which the file was created, last accessed, and last modified, as instances of the class java.nio.file.attribute.FileTime
- Whether the file is a regular file, a directory, a symbolic link, or none of these

- The file size
- The file key—an object of some class, specific to the file system, that may or may not uniquely identify a file

To get these attributes, call

```
BasicFileAttributes attributes = Files.readAttributes(path, BasicFileA
```

If you know that the user's file system is POSIX-compliant, you can instead get an instance of PosixFileAttributes:

```
PosixFileAttributes attributes = Files.readAttributes(path, PosixFileA
```

Then you can find out the group owner and the owner, group, and world access permissions of the file. We won't dwell on the details since so much of this information is not portable across operating systems.

java.nio.file.Files7

- static boolean exists(Path path)
- static boolean isHidden(Path path)
- static boolean isReadable(Path path)
- static boolean isWritable(Path path)
- static boolean isExecutable(Path path)
- static boolean isRegularFile(Path path)
- static boolean isDirectory(Path path)
- static boolean isSymbolicLink(Path path)
  check for the given property of the file given by the path.
- static long size(Path path)
  gets the size of the file in bytes.
- A readAttributes(Path path, Class<A> type, LinkOption... options)
  reads the file attributes of type A.

java.nio.file.attribute.BasicFileAttributes7

- FileTime creationTime()
- FileTime lastAccessTime()

- FileTime lastModifiedTime()
- boolean isRegularFile()
- boolean isDirectory()
- boolean isSymbolicLink()
- boolean isOther()
- long size()
- Object fileKey()
  get the requested attribute.

java.nio.file.attribute.FileTime[7]

- Instant toInstant()[8]
  yields the Instant representing the same time as this FileTime.

2.3.6. Visiting Directory Entries

The static Files.list method returns a Stream<Path> that reads the entries of a directory. The directory is read lazily, making it possible to efficiently process directories with huge numbers of entries.

Since reading a directory involves a system resource that needs to be closed, you should use a try block:

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
   . . .
}
```

The list method does not enter subdirectories. To process all descendants of a directory, use the Files.walk method instead.

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {
   // Contains all descendants, visited in depth-first order
}
```

Here is a sample traversal of the unzipped src.zip tree:

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
. . .
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
. . .
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
. . .
```

As you can see, whenever the traversal yields a directory, it is entered before continuing with its siblings.

You can limit the depth of the tree that you want to visit by calling Files.walk(pathToRoot, depth). Both walk methods have a varargs parameter of type FileVisitOption..., but there is only one option you can supply: FOLLOW_LINKS to follow symbolic links.

> If you filter the paths returned by walk and your filter criterion involves the file attributes stored with a directory, such as size, creation time, or type (file, directory, symbolic link), then use the find method instead of walk. Call that method with a predicate function that accepts a path and a BasicFileAttributes object. The only advantage is efficiency. Since the directory is being read anyway, the attributes are readily available.

This code fragment uses the Files.walk method to copy one directory to another:

```
Files.walk(source).forEach(p -> {
   try {
      Path q = target.resolve(source.relativize(p));
      if (Files.isDirectory(p))
         Files.createDirectory(q);
      else
         Files.copy(p, q);
   }
   catch (IOException e) {
      throw new UncheckedIOException(e);
   }
});
```

Unfortunately, you cannot easily use the Files.walk method to delete a tree of directories since you need to delete the children before deleting the parent. Section 2.3.8 shows you how to overcome that problem.

The Files.walk method throws an exception if any of the subdirectories are not readable. If you only want to visit readable directories, use the walkFileTree method described in the next section.

2.3.7. Glob Patterns

Sometimes, it is useful to filter directory traversals by patterns. The glob syntax can be more useful for this than regular expressions. For example, the glob expression **/*.java denotes Java source files in any nested subdirectory.

Table 2.5 shows all glob patterns.

Pattern	Description	Example
*.java matches all Java files in the current directory.	Matches zero or more characters of a path	

component.	
**	Matches zero or more characters, crossing directory boundaries.	**.java matches all Java files in any subdirectory.
?????.java matches all files with five characters (not counting the extension) and the Java file extension.	Matches a single character.	
Test[0-9A-F].java matches Test*x*.java, where *x* is one hexadecimal digit.	Matches a single characters. You can use hyphens [0-9] and negation [!0-9].	
*.{java,class} matches all Java and class files.	Matches one of the alternatives, separated by commas.	
*** matches all files with a * in their name.	Escapes any of the above as well as \.	

Table 2.5: Glob Patterns

If you use the glob syntax on Windows, you have to escape backslashes twice—once for the glob syntax, and once for the Java string syntax: Files.newDirectoryStream(dir, "C:\\\\").

To filter by such a pattern, construct a PathMatcher and use its matches method.

```
PathMatcher javaSource = FileSystems.getDefault().getPathMatcher(
List<Path> sourcePaths = Files.walk(root).filter(javaSource::matches)
```

java.nio.file.FileSystem 7

- PathMatcher getPathMatcher(String syntaxAndPattern)
  Gets a matcher for the given syntax and pattern. The syntaxAndPattern string starts with a syntax name and a

colon, then the pattern for that syntax. Every file system supports glob and regex syntax.

java.nio.file.FileSystems7

- static FileSystem getDefault()
  gets the default file system.

java.nio.file.PathMatcher7

- boolean matches(Path path)
  returns true if the given path matches.

2.3.8. Directory Streams

As you saw in Section 2.3.6, the Files.walk method produces a Stream<Path> that traverses the descendants of a directory. Sometimes, you need more fine-grained control over the traversal process. In that case, use the Files.newDirectoryStream object instead. It yields a DirectoryStream. Note that this is not a subinterface of java.util.stream.Stream but an interface that is specialized for directory traversal. It is a subinterface of Iterable so that you can use directory stream in an enhanced for loop. Here is the usage pattern:

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir)
   for (Path entry : entries)
      Process entries
}
```

The try-with-resources block ensures that the directory stream is properly closed.

There is no specific order in which the directory entries are visited.

You can filter the files with a glob pattern:

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir,
```

If you want to visit all descendants of a directory, call the walkFileTree method instead and supply an object of type FileVisitor. That object gets notified

- When a file is encountered: FileVisitResult visitFile(T path, BasicFileAttributes attrs)
- Before a directory is processed: FileVisitResult preVisitDirectory(T dir, IOException e)
- After a directory is processed: FileVisitResult postVisitDirectory(T dir, IOException e)
- When an error occurred trying to visit a file or directory, such as trying to open a directory without the necessary permissions: FileVisitResult visitFileFailed(T path, IOException e)

In each case, you can specify whether you want to

- Continue visiting the next file: FileVisitResult.CONTINUE
- Continue the walk, but without visiting the entries in this directory: FileVisitResult.SKIP_SUBTREE
- Continue the walk, but without visiting the siblings of this file: FileVisitResult.SKIP_SIBLINGS
- Terminate the walk: FileVisitResult.TERMINATE

If any of the methods throws an exception, the walk is also terminated, and that exception is thrown from the walkFileTree method.

> The FileVisitor interface is a generic type, but it isn't likely that you'll ever want something other than a FileVisitor<Path>. The walkFileTree method is willing to accept a FileVisitor<? super Path>, but Path does not have an abundance of supertypes.

A convenience class SimpleFileVisitor implements the FileVisitor interface with trivial methods that continue the visit

or terminate it in case of an exception. Extend it and override the methods as needed.

For example, here is how to print out all subdirectories of a given directory:

```
Files.walkFileTree(Path.of("/"), new SimpleFileVisitor<Path>() {
    public FileVisitResult preVisitDirectory(Path path, BasicFileAttri
            throws IOException {
        IO.println(path);
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult postVisitDirectory(Path dir, IOException e)
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult visitFileFailed(Path path, IOException e)
            throws IOException {
        if (e != null) throw e;
        return FileVisitResult.SKIP_SUBTREE;
    }
});
```

Note that we need to override postVisitDirectory and visitFileFailed. Otherwise, the visit would fail as soon as it encounters a directory that it's not allowed to open or a file it's not allowed to access.

Also note that the attributes of the path are passed as an argument to the preVisitDirectory and visitFile methods. The visitor already had to make an OS call to get the attributes, since it needs to distinguish between files and directories. This way, you don't need to make another call.

The other methods of the FileVisitor interface are useful if you need to do some work when entering or leaving a directory. For example, when you delete a directory tree, you need to remove the current directory after you have removed all of its files. Here is the complete code for deleting a directory tree:

```
// Delete the directory tree starting at root
Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
      public FileVisitResult visitFile(Path file, BasicFileAttributes attrs
            throws IOException {
         Files.delete(file);
         return FileVisitResult.CONTINUE;
      }

      public FileVisitResult postVisitDirectory(Path dir, IOException e)
            throws IOException {
         if (e != null) throw e;
         Files.delete(dir);
         return FileVisitResult.CONTINUE;
      }
   });
```

java.nio.file.Files7

- static DirectoryStream<Path> newDirectoryStream(Path path)
- static DirectoryStream<Path> newDirectoryStream(Path path, String glob)
  get an iterator over the files and directories in a given directory. The second method only accepts those entries matching the given glob pattern.
- static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)
  walks all descendants of the given path, applying the visitor to all descendants.

java.nio.file.SimpleFileVisitor<T>7

- static FileVisitResult visitFile(T path, BasicFileAttributes attrs)
  is called when a file or directory is visited; returns one of CONTINUE, SKIP_SUBTREE, SKIP_SIBLINGS, or TERMINATE. The default implementation does nothing and continues.

- static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
- static FileVisitResult postVisitDirectory(T dir, IOException exc)
  are called before and after visiting a directory. The default implementations do nothing and continue. However, if exc is non-null, it is thrown, causing the visit to terminate with that exception.
- static FileVisitResult visitFileFailed(T path, IOException exc)
  is called if an exception was thrown in an attempt to get information about the given file. The default implementation rethrows the exception, which causes the visit to terminate with that exception. Override the method if you want to continue.

2.3.9. ZIP File Systems

The Paths class looks up paths in the default file system—the files on the user's local disk. You can have other file systems. One of the more useful ones is a ZIP file system. If zipname is the name of a ZIP file, then the call

```
FileSystem fs = FileSystems.newFileSystem(zippath);
```

establishes a file system that contains all files in the ZIP archive. It's an easy matter to copy a file out of that archive if you know its name:

```
Files.copy(fs.getPath(sourceName), targetPath);
```

Here, fs.getPath is the analog of Path.of for an arbitrary file system.

To list all files in a ZIP archive, walk the file tree:

```
Files.walk(fs.getPath("/")).forEach(IO::println);
```

To create a new ZIP file, call:

```
FileSystem fs = FileSystems.newFileSystem(zippath, Map.of("create",
```

Then copy files into the file system. When you are done, call

```
fs.close();
```

in order to write the changes to disk.

The ZIP file system is nicer than the API described in Section 2.2.3 which required special classes just to deal with ZIP archives.

java.nio.file.FileSystems7
■ static FileSystem newFileSystem(Path path)13 ■ static FileSystem newFileSystem(Path path, Map<String, ?> env)13 return the file system created by the first file system provider that accepts the given path. By default, there is a provider for ZIP file systems that accepts files whose names end in .zip or .jar. Use the env map for configuration parameters that depend on the file system provider. ■ void close() Closes this file system and permanently writes any changes.

java.nio.file.FileSystem7
■ static Path getPath(String first, String... more) makes a path by joining the given strings.

2.4. Memory-Mapped Files

Most operating systems can take advantage of a virtual memory implementation to “map” a file, or a region of a file, into memory. Then the file can be accessed as if it were an in-

memory array, which is much faster than the traditional file operations.

2.4.1. Memory-Mapped File Performance

At the end of this section, you can find a program that computes the CRC32 checksum of a file using traditional file input and a memory-mapped file. On one machine, we got the timing data shown in Table 2.6 when computing the checksum of the 51MB file src.zip in the lib directory of the JDK.

Method	Time
Plain input stream	11.0 seconds
Buffered input stream	8.8 seconds
RandomAccessFile	162.0 seconds
Memory-mapped file	7.2 seconds

Table 2.6: Timing Data for File Operations

As you can see, on this particular machine, memory mapping is a bit faster than using buffered sequential input and dramatically faster than using a RandomAccessFile.

Of course, the exact values will differ greatly from one machine to another, but it is obvious that the performance gain, compared to random access, can be substantial.

Memory mapping uses a slightly less intuitive API than random access files. Here is what you do.

First, get a channel for the file. A channel is an abstraction for a disk file that lets you access operating system features such as memory mapping, file locking, and fast data transfers between files.

```
FileChannel channel = FileChannel.open(path, options);
```

Then, get a ByteBuffer from the channel by calling the map method of the FileChannel class. Specify the area of the file that you want to map and a mapping mode. Three modes are supported:

- FileChannel.MapMode.READ_ONLY: The resulting buffer is read-only. Any attempt to write to the buffer results in a ReadOnlyBufferException.
- FileChannel.MapMode.READ_WRITE: The resulting buffer is writable, and the changes will be written back to the file at some time. Note that other programs that have mapped the same file might not see those changes immediately. The exact behavior of simultaneous file mapping by multiple programs depends on the operating system.
- FileChannel.MapMode.PRIVATE: The resulting buffer is writable, but any changes are private to this buffer and not propagated to the file.

For example,

```
MappedByteBuffer buffer = channel.map(FileChannel.MapMode.REA
```

Once you have the buffer, you can read and write data using the methods of the ByteBuffer class and the Buffer superclass.

Buffers support both sequential and random data access. A buffer has a position that is advanced by get and put operations. For example, you can sequentially traverse all bytes in the buffer as

```
while (buffer.hasRemaining()) {
   byte b = buffer.get();
   . . .
}
```

Alternatively, you can use random access:

```
for (int i = 0; i < buffer.limit(); i++) {
   byte b = buffer.get(i);
   . . .
}
```

You can also read and write arrays of bytes with the methods

```
get(byte[] bytes)
```

```
get(byte[] bytes, int offset, int length)
```

Finally, there are methods

```
getInt          getChar
getLong           getFloat
getShort          getDouble
```

to read primitive-type values that are stored as binary values in the file. As we already mentioned, Java uses big-endian ordering for binary data. However, if you need to process a file containing binary numbers in little-endian order, simply call

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

To find out the current byte order of a buffer, call

```
ByteOrder b = buffer.order();
```

To write numbers to a buffer, use one of the methods

```
putInt          putChar
putLong           putFloat
putShort          putDouble
```

At some point, and certainly when the channel is closed, these changes are written back to the file.

Listing 2.3 computes the 32-bit cyclic redundancy checksum (CRC32) of a file. That checksum is often used to determine whether a file has been corrupted. Corruption of a file makes it very likely that the checksum has changed. The java.util.zip package contains a class CRC32 that computes the checksum of a sequence of bytes, using the following loop:

```
var crc = new CRC32();
while (more bytes) {
   crc.update(next byte);
}
long checksum = crc.getValue();
```

The details of the CRC computation are not important. We just

use it as an example of a useful file operation. (In practice, you would read and update data in larger blocks, not a byte at a time. Then the speed differences are not as dramatic.)

Run the program as

```
java v2ch02.memoryMap.MemoryMapDemo filename
```

Listing 2.3 **v2ch02/memoryMap/MemoryMapDemo.java**

```
package v2ch02.memoryMap;

import module java.base;

/**
 * This program computes the CRC checksum of a file in four ways.
 */
public class MemoryMapDemo {
   void main(String[] args) throws Exception {
      Path filename = Path.of(args.length > 0 ? args[0]
            : IO.readln("Enter the path to a large file: "));
      IO.println("Input Stream:");
      long start = System.nanoTime();
      long crcValue = checksumInputStream(filename);
      long end = System.nanoTime();
      IO.println(Long.toHexString(crcValue));
      IO.println("%.3f seconds".formatted((end - start) * 1E-9));

      IO.println("Buffered Input Stream:");
      start = System.nanoTime();
      crcValue = checksumBufferedInputStream(filename);
      end = System.nanoTime();
      IO.println(Long.toHexString(crcValue));
      IO.println("%.3f seconds".formatted((end - start) * 1E-9));

      IO.println("Random Access File:");
      start = System.nanoTime();
      crcValue = checksumRandomAccessFile(filename);
      end = System.nanoTime();
      IO.println(Long.toHexString(crcValue));
      IO.println("%.3f seconds".formatted((end - start) * 1E-9));
```

```
    IO.println("Mapped File:");
    start = System.nanoTime();
    crcValue = checksumMappedFile(filename);
    end = System.nanoTime();
    IO.println(Long.toHexString(crcValue));
    IO.println("%.3f seconds".formatted((end - start) * 1E-9));
}

long checksumInputStream(Path filename) throws IOException {
    try (InputStream in = Files.newInputStream(filename)) {
        var crc = new CRC32();
        boolean done = false;
        while (!done) {
            int c = in.read();
            if (c == -1)
                done = true;
            else
                crc.update(c);
        }
        return crc.getValue();
    }
}

long checksumBufferedInputStream(Path filename) throws IOException {
    try (var in = new BufferedInputStream(Files.newInputStream(filename)))
        var crc = new CRC32();

        boolean done = false;
        while (!done) {
            int c = in.read();
            if (c == -1)
                done = true;
            else
                crc.update(c);
        }
        return crc.getValue();
    }
}

long checksumRandomAccessFile(Path filename) throws IOException {
    try (var file = new RandomAccessFile(filename.toFile(), "r")) {
        long length = file.length();
        var crc = new CRC32();
```

```
         for (long p = 0; p < length; p++) {
            int c = file.readByte();
            crc.update(c);
         }
         return crc.getValue();
      }
   }

   long checksumMappedFile(Path filename) throws IOException {
      try (FileChannel channel = FileChannel.open(filename)) {
         var crc = new CRC32();
         int length = (int) channel.size();
         MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_

         for (int p = 0; p < length; p++) {
            int c = buffer.get(p);
            crc.update(c);
         }
         return crc.getValue();
      }
   }
}
```

java.io.FileInputStream1.0

- FileChannel getChannel()1.4
  returns a channel for accessing this input stream.

java.io.FileOutputStream1.0

- FileChannel getChannel()1.4
  returns a channel for accessing this output stream.

java.io.RandomAccessFile1.0

- FileChannel getChannel()1.4
  returns a channel for accessing this file.

java.nio.channels.FileChannel1.4

- static FileChannel open(Path path, OpenOption... options)7
  opens a file channel for the given path. By default, the channel is opened for reading. The parameter options is one of the values WRITE, APPEND,TRUNCATE_EXISTING, CREATE in the StandardOpenOption enumeration.
- MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)
  maps a region of the file to memory. The parameter mode is one of the constants READ_ONLY, READ_WRITE, or PRIVATE in the FileChannel.MapMode class.

java.nio.Buffer1.4

- boolean hasRemaining()
  returns true if the current buffer position has not yet reached the buffer's limit position.
- int limit()
  returns the limit position of the buffer—that is, the first position at which no more values are available.

java.nio.ByteBuffer1.4

- byte get()
  gets a byte from the current position and advances the current position to the next byte.
- byte get(int index)
  gets a byte from the specified index.
- ByteBuffer put(byte b)
  puts a byte at the current position and advances the current position to the next byte. Returns a reference to this buffer.
- ByteBuffer put(int index, byte b)
  puts a byte at the specified index. Returns a reference to this buffer.
- ByteBuffer get(byte[] destination)

- ByteBuffer get(byte[] destination, int offset, int length)
  fill a byte array, or a region of a byte array, with bytes from the buffer, and advance the current position by the number of bytes read. If not enough bytes remain in the buffer, then no bytes are read, and a BufferUnderflowException is thrown. Return a reference to this buffer.
- ByteBuffer put(byte[] source)
- ByteBuffer put(byte[] source, int offset, int length)
  put all bytes from a byte array, or the bytes from a region of a byte array, into the buffer, and advance the current position by the number of bytes read. If not enough bytes remain in the buffer, then no bytes are written, and a BufferOverflowException is thrown. Return a reference to this buffer.
- *Xxx* get*Xxx*()
- *Xxx* get*Xxx*(int index)
- ByteBuffer put*Xxx*(*Xxx* value)
- ByteBuffer put*Xxx*(int index, *Xxx* value)
  get or put a binary number. *Xxx* is one of Int, Long, Short, Char, Float, or Double.
- ByteBuffer order(ByteOrder order)
- ByteOrder order()
  set or get the byte order. The value for order is one of the constants BIG_ENDIAN or LITTLE_ENDIAN of the ByteOrder class.
- static ByteBuffer allocate(int capacity)
  constructs a buffer with the given capacity.
- static ByteBuffer wrap(byte[] values)
  constructs a buffer that is backed by the given array.
- CharBuffer asCharBuffer()
  constructs a character buffer that is backed by this buffer. Changes to the character buffer will show up in this buffer, but the character buffer has its own position, limit, and mark.

java.nio.CharBuffer1.4

- char get()

- CharBuffer get(char[] destination)
- CharBuffer get(char[] destination, int offset, int length)
  get one char value, or a range of char values, starting at the buffer's position and moving the position past the characters that were read. The last two methods return this.
- CharBuffer put(char c)
- CharBuffer put(char[] source)
- CharBuffer put(char[] source, int offset, int length)
- CharBuffer put(String source)
- CharBuffer put(CharBuffer source)
  put one char value, or a range of char values, starting at the buffer's position and advancing the position past the characters that were written. When reading from a CharBuffer, all remaining characters are read. All methods return this.

2.4.2. The Buffer Data Structure

When you use memory mapping, you make a single buffer that spans the entire file or the area of the file that you're interested in. You can also use buffers to read and write more modest chunks of information.

In this section, we briefly describe the basic operations on Buffer objects. A buffer is an array of values of the same type. The Buffer class is an abstract class with concrete subclasses ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, and ShortBuffer.

The StringBuffer class is not related to these buffers.

In practice, you will most commonly use ByteBuffer and CharBuffer. As shown in #ch01fig11, a buffer has

- A capacity that never changes
- A position at which the next value is read or written
- A limit beyond which reading and writing are

meaningless
- Optionally, a mark for repeating a read or write operation

.

Figure 2.5: A buffer

These values fulfill the condition

$$0 \le mark \le position \le limit \le capacity$$

The principal purpose of a buffer is a sequence of “fill, then consume” cycles. At the outset, the buffer’s position is 0 and the limit is the capacity. Keep filling the buffer by calling put or by reading from a channel.

When you run out of data or reach the capacity, it is time to switch to consuming the contents. Call flip to set the limit to the current position and the position to 0.

Now keep consuming, by calling get or writing to a channel, while the remaining method (which computes limit – position) yields a positive value.

When you have consumed all values in the buffer, call compact to prepare the buffer for filling it again. Any unconsumed data is copied to the beginning of the buffer. The position is set to follow the copied data, and the limit is set to the capacity.

If you want to reread the buffer, use rewind or mark/reset (see the API notes for details).

To obtain a buffer, call a static method such as ByteBuffer.allocate or ByteBuffer.wrap.

This loop uses a buffer to transfer data from one channel to another:

```
ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
boolean doneReading = false;
boolean doneWriting = false;
```

```
while (!(doneReading && doneWriting)) {
   if (sourceChannel.read(buffer) == -1) doneReading = true;
   buffer.flip();
   destinationChannel.write(buffer);
   if (buffer.remaining() == 0) doneWriting = true;
   buffer.compact();
}
```

java.nio.Buffer1.4

- Buffer flip()
  prepares this buffer for reading after writing, by setting the limit to the position and the position to 0; returns this.
- Buffer rewind()
  prepares this buffer for rereading the same values by setting the position to 0 and leaving the limit unchanged; returns this.
- Buffer clear()
  prepares this buffer for writing by setting the position to 0 and the limit to the capacity; returns this.
- Buffer mark()
  sets the mark of this buffer to the position; returns this.
- Buffer reset()
  sets the position of this buffer to the mark, thus allowing the marked portion to be read or written again; returns this.
- int remaining()
  returns the remaining number of readable or writable values—that is, the difference between the limit and position.

java.nio.ByteBuffer1.4

- Buffer compact()
  copies any unconsumed bytes to the beginning of the buffer, sets the position to the following byte, and the limit to the capacity; returns this.

java.nio.ReadableByteChannel1.4

- int read(ByteBuffer dst)
  fills the remaining part of the buffer with available bytes, and returns the number of transferred bytes, or -1 if the channel has reached its end.

java.nio.WritableByteChannel1.4

- int write(ByteBuffer src)
  attempts to consume the remaining bytes from the buffer, and returns the number of transferred bytes.

2.5. File Locking

When multiple simultaneously executing programs need to modify the same file, they need to communicate in some way, or the file can easily become damaged. File locks can solve this problem. A file lock controls access to a file or a range of bytes within a file.

Suppose your application saves a file with system-wide preferences. If a user invokes two instances of the application, it could happen that both of them want to write the file at the same time. In that situation, the first instance should lock the file. When the second instance finds the file locked, it can decide to wait until the file is unlocked or simply skip the writing process.

To lock a file, call either the lock or tryLock methods of the FileChannel class.

```
FileChannel channel = FileChannel.open(path);
FileLock lock = channel.lock();
```

or

```
FileLock lock = channel.tryLock();
```

The first call blocks until the lock becomes available. The second call returns immediately, either with the lock or with null if the lock is not available. The file remains locked until the channel is closed or the release method is invoked on the lock.

You can also lock a portion of the file with the call

```
FileLock lock(long start, long size, boolean shared)
```

or

```
FileLock tryLock(long start, long size, boolean shared)
```

The shared flag is false to lock the file for both reading and writing. It is true for a shared lock, which allows multiple processes to read from the file, while preventing any process from acquiring an exclusive lock. Not all operating systems support shared locks. You may get an exclusive lock even if you just asked for a shared one. Call the isShared method of the FileLock class to find out which kind you have.

If you lock the tail portion of a file and the file subsequently grows beyond the locked portion, the additional area is not locked. To lock all bytes, use a size of Long.MAX_VALUE.

Be sure to unlock the lock when you are done. As always, this is best done with a try-with-resources statement:

```
try (FileLock _ = channel.lock()) {
   access the locked file or segment
}
```

Keep in mind that file locking is system-dependent. Here are some points to watch for:

- On some systems, including Linux, file locking is merely advisory. If an application fails to get a lock, it may still write to a file that another application has currently

locked.
- On some systems, you cannot simultaneously lock a file and map it into memory.
- File locks are global to the Java virtual machine. Concurrent threads can't each acquire a lock on the same file. The lock and tryLock methods will throw an OverlappingFileLockException if another thread already holds a lock on the same file or an overlapping range.
- On some systems, closing a channel releases all locks on the underlying file. You should therefore avoid multiple channels on the same locked file.
- Locking files on a networked file system is highly system-dependent and should probably be avoided.
- File locking is only guaranteed for Java programs. Programs written in other languages may use different and incompatible locking mechanisms.

The program in Listing 2.4 shows a simple example of file locking. Multiple users want to register their passwords in a common file. The program reads in the file, adds the username and hashed password, and writes the resulting file. It is important that this update is atomic. Otherwise, the password file can be corrupted if two users update it concurrently.

See Chapter 9 for more information about password hashing. It has nothing to do with file locking, but I didn't want to show a sample program that saves unhashed passwords.

The program shows the limitations of the file locking API. It is tied to the FileChannel API, which is based on the reading and writing of byte buffers. The program has to turn those byte buffers into readers and writers.

An alternative would be to use a RandomAccessFile whose getChannel method yields a channel to the file. But it is equally inconvenient. One needs to read the file into a byte array and prepare a byte array to write the modified file.

Listing 2.4 **v2ch02/fileLock/FileLockDemo.java**

```
package v2ch02.fileLock;

import module java.base;

/**
 * This program writes user names and passwords to the file passwords.prope
 * atomically updated. With Linux, try
 * for f in $(seq 1 100) ; do (java fileLock.FileLockDemo user$f secret$f &) ; don
 * When all Java processes have finished, then passwords.properties
 * has an entry for each of them.
 */

public class FileLockDemo {
   void main(String[] args) throws Exception {
      Path path = Path.of("password.properties");
      FileChannel channel = FileChannel.open(path, StandardOpenOption.READ
            StandardOpenOption.WRITE, StandardOpenOption.CREATE);
      try (FileLock _ = channel.lock()) {
         // Read bytes from file
         ByteBuffer readBuffer = ByteBuffer.allocate((int) channel.size());
         channel.read(readBuffer);

         // Read props from bytes
         Reader in = new StringReader(new String(readBuffer.array()));
         Properties props = new Properties();
         props.load(in);

         // Get username, password from args or user input
         String username;
         char[] password;
         if (args.length >= 2) {
            username = args[0];
            password = args[1].toCharArray();
         }
         else {
            Console console = System.console();
            username = console.readLine("User name: ");
            password = console.readPassword("Password: ");
         }

         // Hash password
         final int ITERATIONS = 210_000;
         final int SALT_BYTES = 16;
         final int HASH_BYTES = 32;
```

```
            SecureRandom random = new SecureRandom();
            byte[] salt = new byte[SALT_BYTES];
            random.nextBytes(salt);

            var spec = new PBEKeySpec(password, salt, ITERATIONS, 8 * HASH_BYT
            SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHma
            byte[] hash = skf.generateSecret(spec).getEncoded();
            Base64.Encoder encoder = Base64.getEncoder();

            // Put username and hashed password into props
            props.put(username, ITERATIONS + "|" + encoder.encodeToString(salt) +
                  + encoder.encodeToString(hash));

            // Write props to bytes
            StringWriter out = new StringWriter();
            props.store(out, null);
            byte[] outBytes = out.toString().getBytes();

            // Write bytes to file
            channel.truncate(0);
            ByteBuffer writeBuffer = ByteBuffer.wrap(outBytes);
            channel.write(writeBuffer);
         }
      }
   }
```

java.nio.channels.FileChannel 1.4

- FileLock lock()
  acquires an exclusive lock on the entire file. This method blocks until the lock is acquired.
- FileLock tryLock()
  acquires an exclusive lock on the entire file, or returns null if the lock cannot be acquired.
- FileLock lock(long position, long size, boolean shared)
- FileLock tryLock(long position, long size, boolean shared)
  acquire a lock on a region of the file. The first method blocks until the lock is acquired, and the second method returns null if the lock cannot be acquired. The parameter shared is true for a shared lock, false for an exclusive lock.
- FileChannel truncate(long size)
  truncates the file to the given size if it is larger.

java.nio.channels.FileLock1.4

- void close()7
  releases this lock.

java.io.RandomAccessFile1.0

- FileChannel getChannel()1.4
  yields the channel for this random access file.

2.6. Object Input/Output Streams and Serialization

In Section 2.1.7 and Section 2.2.2, you have seen how to store Employee objects in text or binary format. However, objects that you create in an object-oriented program are rarely all of the same type. For example, you might have an array called staff that is nominally an array of Employee records but contains objects that are actually instances of a subclass such as Manager.

It is certainly possible to come up with a data format that allows you to store such polymorphic collections—but, fortunately, we don't have to. The Java language supports a very general mechanism, called object serialization, that makes it possible to write any object to an output stream and read it again later. (You will see in the following sections where the term "serialization" comes from.)

2.6.1. Saving and Loading Serializable Objects

To save object data, you first need to open an ObjectOutputStream object:

```
var out = new ObjectOutputStream(new FileOutputStream("employe
```

Now, to save an object, simply use the writeObject method of the ObjectOutputStream class as in the following fragment:

```
var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
var boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
out.writeObject(harry);
out.writeObject(boss);
```

To read the objects back in, first get an ObjectInputStream object:

```
var in = new ObjectInputStream(new FileInputStream("employee.da
```

Then, retrieve the objects in the same order in which they were written, using the readObject method:

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

There is, however, one change you need to make to any class that you want to save to an output stream and restore from an object input stream. The class must implement the Serializable interface:

```
class Employee implements Serializable { . . . }
```

The Serializable interface has no methods, so you don't need to change your classes in any way. In this regard, it is similar to the Cloneable interface that we discussed in Chapter 6 of Volume I. However, to make a class cloneable, you still had to override the clone method of the Object class. To make a class serializable, you do not need to do anything else.

You can write and read only objects with the writeObject/readObject methods. For primitive type values, use methods such as writeInt/readInt or writeDouble/readDouble. (The object input/output stream classes implement the DataInput/DataOutput interfaces.)

Behind the scenes, an ObjectOutputStream looks at all the fields

of the objects and saves their contents. For example, when writing an Employee object, the name, date, and salary fields are written to the output stream.

However, there is one important situation to consider: What happens when one object is shared by several objects as part of their state?

To illustrate the problem, let us make a slight modification to the Manager class. Let's assume, nostalgically, that each manager has a secretary:

```
class Manager extends Employee {
   private Employee secretary;
   . . .
}
```

Each Manager object now contains a reference to an Employee object that describes the secretary. Of course, two managers can share the same secretary, as is the case in #ch01fig06 and the following code:

```
var harry = new Employee("Harry Hacker", . . .);
var carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
var tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```

.

Figure 2.6: Two managers can share a mutual employee.

Saving such a network of objects is a challenge. Of course, we cannot save and restore the memory addresses for the secretary objects. When an object is reloaded, it will likely occupy a completely different memory address than it originally did.

Instead, each object is saved with a serial number—hence the name object serialization for this mechanism. Here is the

algorithm:

1. Associate a serial number with each object reference that you encounter (as shown in #ch01fig07).

.

Figure 2.7: An example of object serialization

2. When encountering an object reference for the first time, save the object data to the output stream.
3. If it has been saved previously, just write “same as the previously saved object with serial number *x*.”

When reading the objects back, the procedure is reversed.

1. When an object is specified in an object input stream for the first time, construct it, initialize it with the stream data, and remember the association between the serial number and the object reference.
2. When the tag “same as the previously saved object with serial number *x*” is encountered, retrieve the object reference for the sequence number.

In this chapter, we will use serialization to save a collection of objects to a disk file and retrieve it exactly as we stored it. Another very important application is the transmittal of a collection of objects across a network connection to another computer. Just as raw memory addresses are meaningless in a file, they are also meaningless when you communicate with a different processor. By replacing memory addresses with serial numbers, serialization permits the transport of object collections from one machine to another.

If the superclass of a serializable class is not serializable, it must have an accessible no-argument constructor. Consider this example:

```
class Person // Not serializable
class Employee extends Person implements Serializable
```

When an Employee object is deserialized, its instance fields are read from the object input stream, but the Person instance fields are set by the Person constructor.

Listing 2.5 is a program that saves and reloads a network of Employee and Manager objects (some of which share the same employee as a secretary). Note that the secretary object is unique after reloading—when newStaff[1] gets a raise, that is reflected in the secretary fields of the managers.

Listing 2.5 **v2ch02/serial/ObjectStreamDemo.java**

```
package v2ch02.serial;

import module java.base;

class ObjectStreamDemo {
   void main() throws Exception {
      var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      var carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      carl.setSecretary(harry);
      var tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
      tony.setSecretary(harry);

      var staff = new Employee[3];

      staff[0] = carl;
      staff[1] = harry;
      staff[2] = tony;

      // save all employee records to the file employee.ser
      try (var out = new ObjectOutputStream(new FileOutputStream("employee
         out.writeObject(staff);
      }

      try (var in = new ObjectInputStream(new FileInputStream("employee.ser"
         // retrieve all records into a new array
```

```
         var newStaff = (Employee[]) in.readObject();

         // raise secretary's salary
         newStaff[1].raiseSalary(10);

         // print the newly read employee records
         for (Employee e : newStaff)
            IO.println(e);
      }
   }
}
```

java.io.ObjectOutputStream1.1

- ObjectOutputStream(OutputStream out)
  creates an ObjectOutputStream so you can write objects to the specified OutputStream.
- void writeObject(Object obj)
  writes the specified object to the ObjectOutputStream. This method saves the class of the object, the signature of the class, and the values of any nonstatic, nontransient fields of the class and its superclasses.

java.io.ObjectInputStream1.1

- ObjectInputStream(InputStream in)
  creates an ObjectInputStream to read back object information from the specified InputStream.
- Object readObject()
  reads an object from the ObjectInputStream. In particular, this method reads back the class of the object, the signature of the class, and the values of the nontransient and nonstatic fields of the class and all its superclasses. It does deserializing so that multiple object references can be recovered.

2.6.2. Understanding the Object Serialization File Format

Object serialization saves object data in a particular file format. Of course, you can use the writeObject/readObject methods without having to know the exact sequence of bytes that represents objects in a file. Nonetheless, we found studying the data format extremely helpful for gaining insight into the object serialization process. As the details are somewhat technical, feel free to skip this section if you are not interested in the implementation.

Every file begins with the two-byte “magic number”

```
AC ED
```

followed by the version number of the object serialization format, which is currently

```
00 05
```

(We use hexadecimal numbers throughout this section to denote bytes.) Then, it contains a sequence of objects, in the order in which they were saved.

String objects are saved as

- 74
- Two-byte length
- Characters in the “modified UTF-8” format

For example, the string “Harry” is saved as

```
74 00 05 Harry
```

When an object is saved, the class of that object must be saved as well. The class descriptor has the following format:

- 72
- 2-byte length of class name
- Class name
- 8-byte fingerprint
- 1-byte flag
- 2-byte count of instance field descriptors
- Instance field descriptors

- 78 (end marker)
- Superclass descriptor (70 if none)

The fingerprint is obtained by ordering the descriptions of the class, superclass, interfaces, field types, and method signatures in a canonical way, and then applying the Secure Hash Algorithm version 1 (SHA-1) to that data.

SHA-1 gives a “fingerprint” of a block of information. This fingerprint is always a 20-byte data packet, regardless of the size of the original data. It is created by a clever sequence of bit operations on the data that makes it extremely likely that the fingerprint will change if the information is altered in any way. The serialization mechanism uses only the first eight bytes of the SHA-1 code as a class fingerprint. It is still very likely that the class fingerprint will change if the instance fields or methods change.

When reading an object, its fingerprint is compared against the current fingerprint of the class. If they don’t match, it means the class definition has changed after the object was written, and an exception is generated. Of course, in practice, classes do evolve, and it might be necessary for a program to read in older versions of objects. We will discuss this in Section 2.6.7.

The flag byte is composed of the bit masks, defined in java.io.ObjectStreamConstants, shown in Table 2.7.

Name	Value	Description
SC_WRITE_METHOD	1	Class defines writeObject method.
SC_SERIALIZABLE	2	Class implements Serializable.
SC_EXTERNALIZABLE	4	Class implements Externalizable.
SC_BLOCK_DATA	8	Opaque data written as “block data” (default since JDK 1.2).
SC_ENUM	16	Class is an

		enumeration.

Table 2.7: Bit Masks

The classes that we write implement the Serializable interface and will have a flag value of 02. The serializable java.util.Date class has a flag of 03 because it defines its own writeObject method (see Section 2.6.4). We discuss the Externalizable interface in Section 2.6.5.

Each instance field descriptor has the format:

- 1-byte type code
- 2-byte length of field name
- Field name
- Class name (if the field is an object)

The type code is one of the following:

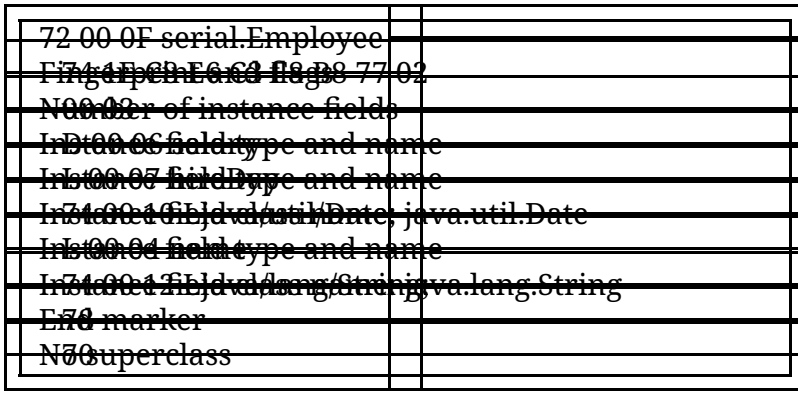
B	Byte
C	Char
D	Double
F	Float
I	Int
J	Long
L	Object
S	Short
Z	Boolean
[	Array

When the type code is L, the field name is followed by the field type. Class and field name strings do not start with the string code 74, but field types do. Field types use a slightly different encoding of their names—namely, the format used by native methods.

For example, the salary field of the Employee class is encoded as

```
D 00 06 salary
```

Here is the complete class descriptor of the Employee class, contained in the serial package in our example program:


72 00 0F serial.Employee
Fingerprint and flags
Number of instance fields
Instance field type and name
Instance field type and name
Instance field type and name
Instance field type and name
74 00 10 java/util/Date java.util.Date
74 00 12 java/lang/String java.lang.String
78 marker
70 superclass

These descriptors are fairly long. If the same class descriptor is needed again in the file, an abbreviated form is used:

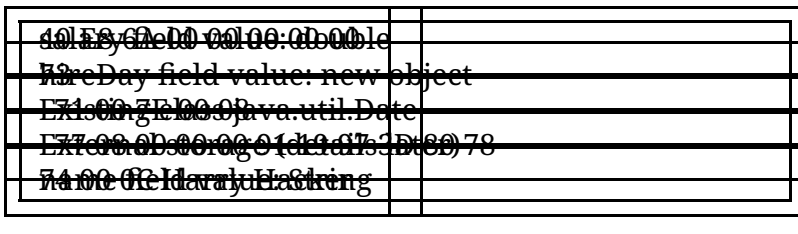
- 71
- 4-byte serial number

The serial number refers to the previous explicit class descriptor. We discuss the numbering scheme later.

An object is stored as

- 73
- Class descriptor
- Object data

For example, here is how the instance fields of an Employee object are stored:



As you can see, the data file contains enough information to restore the Employee object. (In this example, we store the hireday as a Date instead of a LocalDate, since the LocalDate

class has a more complex serialization.)

As you just saw, the object stream contains the names of the classes, superclasses, and fields of all serialized objects. For inner classes, some of these names are synthesized by the compiler, and the naming convention may change from one Java implementation to another. If that occurs, deserialization fails. Therefore, there is a small risk with serializing inner classes if you intend to have them deserialized with a different implementation.

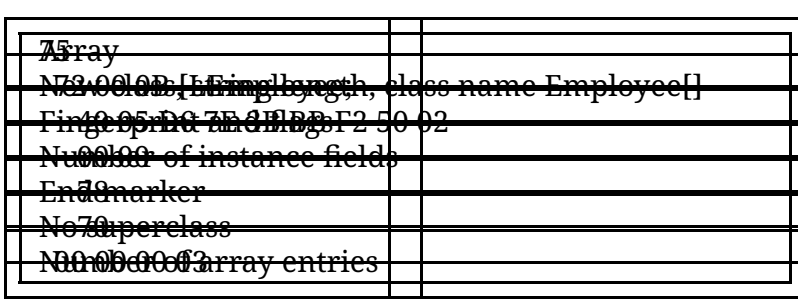
Static inner classes, including inner enumerations and records, are safe to serialize.

Arrays are saved in the following format:

- 75
- Class descriptor
- 4-byte number of elements
- Elements

The array class name in the class descriptor is in the same format as that used by native methods (which is slightly different from the format used by class names in other class descriptors). In this format, class names start with an L and end with a semicolon.

For example, an array of three Employee objects starts out like this:



Note that the fingerprint for an array of Employee objects is different from a fingerprint of the Employee class itself.

All objects (including arrays and strings) and all class descriptors are given serial numbers as they are saved in the output file. The numbers start at 00 7E 00 00.

We already saw that a full class descriptor for any given class occurs only once. Subsequent descriptors refer to it. For example, in our previous example, a repeated reference to the Date class was coded as

```
71 00 7E 00 08
```

The same mechanism is used for objects. If a reference to a previously saved object is written, it is saved in exactly the same way—that is, 71 followed by the serial number. It is always clear from the context whether a particular serial reference denotes a class descriptor or an object.

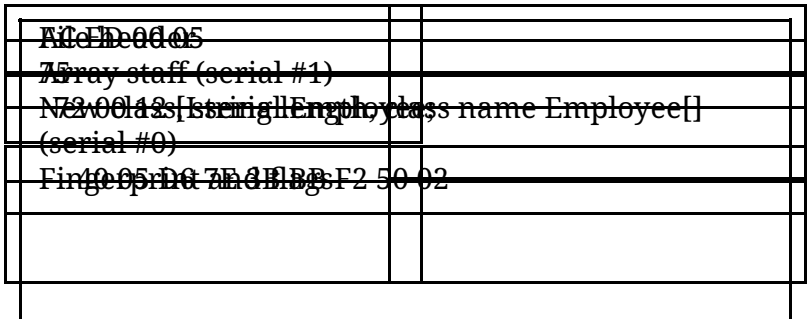
Some classes (such as the Date class) store object state in an opaque binary block, encoded as

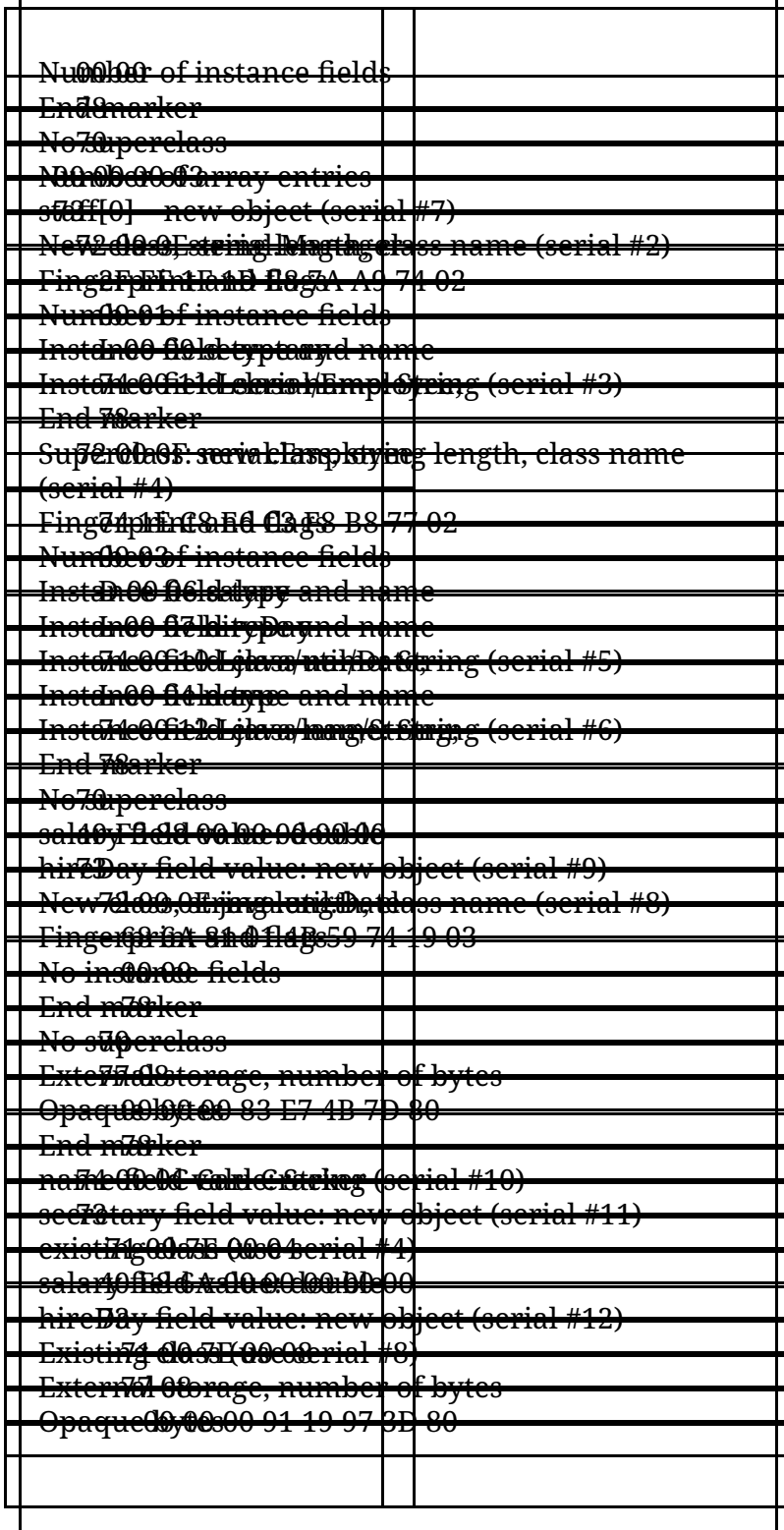
- 77 (short block) or 7A (long block)
- 1-byte or 4-byte block length
- Block contents
- 78

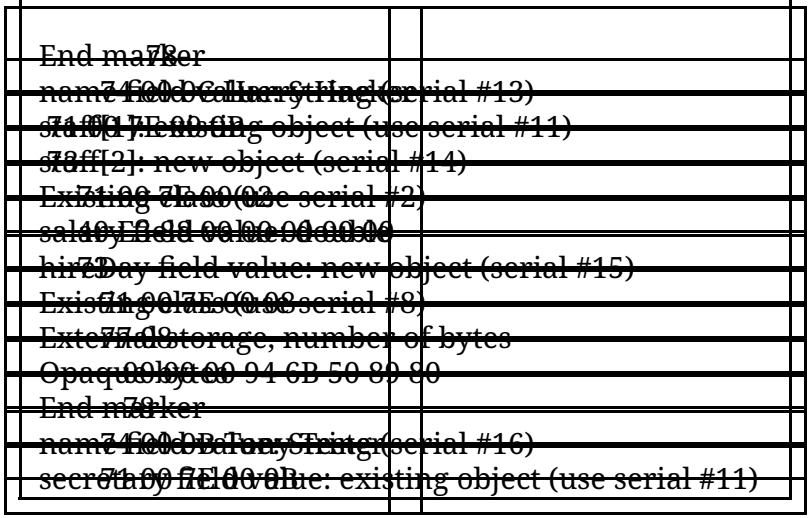
Finally, a null reference is stored as

```
70
```

Here is the commented output of the ObjectStreamDemo program of the preceding section. Run the program, look at a hex dump of its data file employee.ser, and compare it with the commented listing. The most important lines are toward the end of the output. They show a reference to a previously saved object.







Of course, studying these codes can be about as exciting as reading a phone book. It is not important to know the exact file format (unless you are trying to create an evil effect by modifying the data), but it is still instructive to know that the serialized format has a detailed description of all the objects it contains, with sufficient detail to allow reconstruction of both objects and arrays of objects.

What you should remember is this:

- The serialized format contains the types and instance fields of all objects.
- Each object is assigned a serial number.
- Repeated occurrences of the same object are stored as references to that serial number.

2.6.3. Transient Fields

Certain fields should never be serialized—for example, database connections that are meaningless when an object is reconstituted. Also, when an object keeps a cache of values, it might be better to drop the cache and recompute it instead of storing it.

To prevent a field from being serialized, tag it with the transient modifier. Always mark fields as transient if they hold

instances of nonserializable classes. Transient fields are simply skipped when objects are serialized.

2.6.4. The readObject and writeObject Methods

In rare cases, you need to tweak the serialization mechanism. A serializable class can add any desired action to the default read and write behavior by defining methods with the signature

```
@Serial private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException;
@Serial private void writeObject(ObjectOutputStream out)
      throws IOException;
```

Then, the object headers continue to be written as usual, but the fields are no longer automatically serialized. Instead, these methods are called.

Note the @Serial annotation. The methods for tweaking serialization don't belong to interfaces. Therefore, you can't use the @Override annotation to have the compiler check the method declarations. The @Serial annotation is meant to enable the same checking for serialization methods. Up to Java 21, the javac compiler doesn't do such checking, but that might change in the future. The IntelliJ IDE uses the annotation.

Here is a typical example for customization. A number of classes in the java.awt.geom package, such as Point2D.Double, are not serializable. Now, suppose you want to serialize a class LabeledPoint that stores a String and a Point2D.Double. First, you need to mark the Point2D.Double field as transient to avoid a NotSerializableException.

```
public class LabeledPoint implements Serializable {
   private String label;
   private transient Point2D.Double point;
   . . .
}
```

In the writeObject method, first write the object descriptor and the String field, label, by calling the defaultWriteObject method.

This is a special method of the ObjectOutputStream class that can only be called from within a writeObject method of a serializable class. Then we write the point coordinates, using the standard DataOutput calls.

```
private void writeObject(ObjectOutputStream out) throws IOExcepti
   out.defaultWriteObject();
   out.writeDouble(point.getX());
   out.writeDouble(point.getY());
}
```

In the readObject method, we reverse the process:

```
private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException {
   in.defaultReadObject();
   double x = in.readDouble();
   double y = in.readDouble();
   point = new Point2D.Double(x, y);
}
```

Another example is the HashSet class that supplies its own readObject and writeObject methods. Instead of saving the internal structure of the hash table, the writeObject method simply saves the capacity, load factor, size, and elements. The readObject method reads back the capacity and load factor, constructs a new table, and inserts the elements.

The readObject and writeObject methods only need to save and load their data. They do not concern themselves with superclass data or any other class information.

The Date class uses this approach. Its writeObject method saves the milliseconds since the “epoch” (January 1, 1970). The data structure that caches calendar data is not saved. For that reason, the serialized Date instances in Section 2.6.2 consist of an opaque data block.

Just like a constructor, the readObject method operates on partially initialized objects. If you call a non-final

---

method inside readObject that is overridden in a subclass, it may access uninitialized data.

If a serializable class defines a field

```
@Serial private static final ObjectStreamField[] serialPersister
```

then serialization uses those field descriptors instead of the non-transient non-static fields. There is also an API for setting the field values before serialization, or reading them after deserialization. This is useful for preserving a legacy layout after a class has evolved. For example, the BigDecimal class uses this mechanism to serialize its instances in a format that no longer reflects the instance fields.

2.6.5. The readExternal and writeExternal Methods

Instead of letting the serialization mechanism save and restore object data, a class can define its own mechanism. For example, you can encrypt the data or use a format that is more efficient than the serialization format.

To do this, a class must implement the Externalizable interface. This, in turn, requires it to define two methods:

```
public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutput out) throws IOException;
```

Unlike the readObject and writeObject methods, these methods are fully responsible for saving and restoring the entire object, including the superclass data. When writing an object, the serialization mechanism merely records the class of the object in the output stream. When reading an externalizable object, the object input stream creates an object with the no-argument constructor and then calls the readExternal method.

In this example, the LabeledPixel class extends the serializable Point class, but it takes over the serialization of the class and

superclass. The fields of the object are not stored in the standard serialization format. Instead, the data are placed in an opaque block.

```
public class LabeledPixel extends Point implements Externalizable {
   private String label;

   public LabeledPixel() {} // required for externalizable class

   public void writeExternal(ObjectOutput out)
         throws IOException {
      out.writeInt((int) getX());
      out.writeInt((int) getY());
     out.writeUTF(label);
   }

   public void readExternal(ObjectInput in)
         throws IOException, ClassNotFoundException {
     int x = in.readInt();
     int y = in.readInt();
     setLocation(x, y);
     label = in.readUTF();
   }
   . . .
}
```

The readExternal and writeExternal methods should not be annotated with @Serial. Since they are defined in the Externalizable interface, you can simply annotate them with @Override.

Unlike the readObject and writeObject methods, which are private and can only be called by the serialization mechanism, the readExternal and writeExternal methods are public. In particular, readExternal potentially permits modification of the state of an existing object.

2.6.6. The readResolve and writeReplace Methods

We take it for granted that objects can only be constructed with the constructor. However, a deserialized object is not constructed. Its fields are simply restored from an object stream.

This is a problem if the constructor enforces some condition. For example, a singleton object may be implemented so that the constructor can only be called once. As another example, database entities can be constructed so that they always come from a pool of managed instances.

You shouldn't implement your own mechanism for singletons. If you need a singleton, make an enumerated type with one instance which is, by convention, called INSTANCE.

```
public enum PersonDatabase {
   INSTANCE;

   public Person findById(int id) { . . . }
   . . .
}
```

This works because enum instances are guaranteed to be deserialized properly.

Now let's suppose that you are in the rare situation where you want to control the identity of each deserialized instance. As an example, suppose a Person class wants to restore its instances from a database when deserializing. Then you should not serialize the object itself. Instead, request that a proxy instance is saved. When restored, that proxy locates and constructs the desired object. Your class needs to provide a writeReplace method that returns the proxy object:

```
public class Person implements Serializable {
   private int id;
   // Other fields
   . . .
   @Serial private Object writeReplace() {
```

```
        return new PersonProxy(id);
    }
}
```

When a Person object is serialized, none of its fields are saved. Instead, the writeReplace method is called and its return value is serialized and written to the stream.

The proxy class needs to implement a readResolve method that yields a Person instance:

```
class PersonProxy implements Serializable {
    private int id;

    public PersonProxy(int id) {
        this.id = id;
    }

    @Serial private Object readResolve() {
        return PersonDatabase.INSTANCE.findById(id);
    }
}
```

When the readObject method finds a PersonProxy in an ObjectInputStream, it deserializes the proxy, calls its readResolve method, and returns the result.

Unlike the readObject and writeObject methods, the readResolve and writeReplace methods need not be private.

With enumerations and records, readObject/writeObject or readExternal/writeExternal methods are not used for serialization. With records, but not with enumerations, the writeReplace method will be used.

2.6.7. Versioning

If you use serialization to save objects, you need to consider what happens when your program evolves. Can version 1.1 read the old files? Can the users who still use 1.0 read the files that the new version is producing? Clearly, it would be desirable if object files could cope with the evolution of classes.

At first glance, it seems that this would not be possible. When a class definition changes in any way, its SHA fingerprint also changes, and you know that object input streams will refuse to read in objects with different fingerprints. However, a class can indicate that it is compatible with an earlier version of itself. To do this, you must first obtain the fingerprint of the earlier version of the class. Use the standalone serialver program that is part of the JDK to obtain this number. For example, running

```
serialver serial.Employee
```

prints

```
serial.Employee: static final long serialVersionUID = 8367346051156
```

All later versions of the class must define the serialVersionUID constant to the same fingerprint as the original.

```
class Employee implements Serializable { // version 1.1
   . . .
   @Serial public static final long serialVersionUID = 8367346051156
}
```

When a class has a static data member named serialVersionUID, it will not compute the fingerprint manually but will use that value instead.

Once that static data member has been placed inside a class, the serialization system is now willing to read in different versions of objects of that class.

If only the methods of the class change, there is no problem with reading the new object data. However, if the instance fields change, you may have problems. For example, the old file object may have more or fewer instance fields than the one in

the program, or the types of the instance fields may be different. In that case, the object input stream makes an effort to convert the serialized object to the current version of the class.

The object input stream compares the instance fields of the current version of the class with those of the version in the serialized object. Of course, the object input stream considers only the nontransient and nonstatic instance fields. If two fields have matching names but different types, the object input stream makes no effort to convert one type to the other—the objects are incompatible. If the serialized object has instance fields that are not present in the current version, the object input stream ignores the additional data. If the current version has instance fields that are not present in the serialized object, the added fields are set to their default (null for objects, zero for numbers, and false for boolean values).

Here is an example. Suppose we have saved a number of employee records on disk, using the original version (1.0) of the class. Now we change the Employee class to version 2.0 by adding a instance field called department. #ch01fig09 shows what happens when a 1.0 object is read into a program that uses 2.0 objects. The department field is set to null. #ch01fig10 shows the opposite scenario: A program using 1.0 objects reads a 2.0 object. The additional department field is ignored.

.

Figure 2.8: Reading an object with fewer instance fields

.

Figure 2.9: Reading an object with more instance fields

Is this process safe? It depends. Dropping a instance field seems harmless—the recipient still has all the data that it knows how to manipulate. Setting a instance field to null might not be so safe. Many classes work hard to initialize all instance fields in all constructors to non-null values, so that the methods don't

have to be prepared to handle null data. It is up to the class designer to implement additional code in the readObject method to fix version incompatibilities or to make sure the methods are robust enough to handle null data.

Before you add a serialVersionUID field to a class, ask yourself why you made your class serializable. If serialization is used only for short-term persistence, such as distributed method calls in an application server, there is no need to worry about versioning and the serialVersionUID. The same applies if you extend a class that happens to be serializable, but you have no intent to ever persist its instances. If your IDE gives you pesky warnings, change the IDE preferences to turn them off, or add an annotation @SuppressWarnings("serial"). This is safer than adding a serialVersionUID that you may later forget to change.

Enumerations and records ignore the serialVersionUID field. An enumeration always has a serialVersionUID of 0L. You can declare the serialVersionUID of a record, but the IDs don't have to match for deserialization.

In this section, you saw what happens when the reader's version of a class has instance fields that aren't present in the object stream. It is also possible during class evolution for a superclass to be added. Then a reader using the new version may read an object stream in which the instance fields of the superclass are not set. By default, those instance fields are set to their zero/null default. That may leave the superclass in an unsafe state. The superclass can defend against that problem by defining an initialization method

```
@Serial private void readObjectNoData() throws ObjectStream
```

The method should either set the same state as the no-

argument constructor or throw an InvalidObjectException. It is only called in the unusual circumstance that an object stream is read which contains an instance of a subclass with missing superclass data.

2.6.8. Deserialization and Security

During deserialization of a serializable class, objects are created without invoking any constructor of the class. Even if the class has a no-argument constructor, it is not used. The field values are set directly from the values of the object input stream.

For serializable records, deserialization calls the canonical constructor, passing it the values of the components from the object input stream. (As a consequence, cyclic references in records are not restored.)

Bypassing constructors is a security risk. An attacker can craft bytes describing an invalid object that could have never been constructed. Suppose, for example, that the Employee constructor throws an exception when called with a negative salary. We would like to think that this means that no Employee object can have a negative salary. But, as you saw in the preceding section, it is not difficult to inspect the bytes for a serialized object and modify some of them. By doing so, one can craft bytes for an employee with a negative salary and then deserialize them.

A serializable class can optionally implement the ObjectInputValidation interface and define a validateObject method to check whether its objects are properly deserialized. For example, the Employee class can check that salaries are not negative:

```
public void validateObject() throws InvalidObjectException {
   IO.println("validateObject");
```

```
    if (salary < 0)
      throw new InvalidObjectException("salary < 0");
  }
```

Unfortunately, the method is not invoked automatically. To invoke it, you must also provide the following method:

```
  @Serial private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
    in.registerValidation(this, 0);
    in.defaultReadObject();
  }
```

The object is then scheduled for validation, and the validateObject method is called when this object and all dependent objects have been loaded. The second parameter lets you specify a priority. Validation requests with higher priorities are done first.

There are other security risks. Adversaries can create data structures that consume enough resources to crash a virtual machine. More insidiously, any class on the class path can be deserialized. Hackers have been devious about piecing together "gadget chains", sequences of operations in various utility classes that use reflection and culminate in calling methods such as Runtime.exec with a string of their choice.

Any application that receives serialized data from untrusted sources over a network connection is vulnerable to such attacks. For example, some servers serialize session data and deserialize whatever data is returned in the HTTP session cookie.

You should avoid situations in which arbitrary data from untrusted sources are deserialized. In the example of session data, the server should sign the data, and only deserialize data with a valid signature.

JEP 290 and 415 provide a serialization filter mechanism to harden applications from such attacks. The filters can see the names of deserialized classes and several metrics (stream size,

array sizes, total number of references, longest chain of references). Based on those data, the deserialization can be aborted.

In its simplest form, you provide a pattern describing the valid and invalid classes. For example, if you start our sample serialization demo as

```
java -Djdk.serialFilter='serial.*;java.**;!*' v2ch02.serial.ObjectStream
```

then the objects will be loaded. The filter allows all classes in the serial package and all classes whose package name starts with java, but no others. If you don't allow java.**, or at least java.util.Date, deserialization fails.

You can place the filter pattern into a configuration file and specify multiple filters for different purposes. You can also implement your own filters. See https://docs.oracle.com/en/java/javase/21/core/serialization-filtering1.html for details.

2.6.9. Using Serialization for Cloning

There is an amusing use for the serialization mechanism: It gives you an easy way to clone an object, provided the class is serializable. Simply serialize it to an output stream and then read it back in. The result is a new object that is a deep copy of the existing object. You don’t have to write the object to a file—you can use a ByteArrayOutputStream to save the data into a byte array.

As Listing 2.6 shows, to get clone for free, simply extend the SerialCloneable class, and you are done.

You should be aware that this method, although clever, will usually be much slower than a clone method that explicitly constructs a new object and copies or clones the instance fields.

Listing 2.6 **v2ch02/serialClone/SerialCloneDemo.java**

```
package v2ch02.serialClone;

class SerialCloneDemo {
   void main() throws Exception {
      var harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
      // clone harry
      var harry2 = (Employee) harry.clone();

      // mutate harry
      harry.raiseSalary(10);

      // now harry and the clone are different
      IO.println(harry);
      IO.println(harry2);
   }
}
```

Serialization attempts to restore cyclic references between objects, but this can lead to problems. Consider the following situation:

```
var a = new ArrayList<Object>();
var s = new HashSet<Object>();
a.add(s);
s.add(a); // Cyclic references between a and s
IO.println(SerialCloneable.serialClone(s).size()); // 1
IO.println(SerialCloneable.serialClone(a).size()); // NullPointer
```

For convenience, this example uses the serialClone method from the sample code that serializes an object, and then deserializes it again.

Why does the second deserialization go awry? Array lists and hash sets do not serialize their internal data structure. Instead, their writeObject methods write out the serialized elements, and their readObject methods insert the deserialized elements into a new array or hash set. When a is deserialized in the last line, the following series of unfortunate events occurs:

1. The size field of the array list is restored from the

stream, but elements is still null
2. A temporary array is allocated that later will be assigned to the elements field of the array list
3. Deserializing of the array elements commences. There is just one, the hash set
4. The hash set deserializes its sole element, the array list, obtaining a reference to the partially deserialized object
5. The hash set attempts to compute the hash code of the array list
6. The ArrayList.hashCode method throws a NullPointerException since the elements field is null

As you can see, the deserialization strategy for ArrayList is fundamentally flawed. It should not be possible to observe a partially constructed instance.

The deserialization strategy for HashSet has a different flaw, calling hashCode on elements that may only be partially deserialized.

In hindsight, serialization in Java has not been a success. It is complex and error-prone, and provides a significant attack surface. It is good to understand the mechanism, and its limitations. But you should think twice about using it for anything other than short-term persistence or secured communication between virtual machines of the same type.

2.7. Regular Expressions

Regular expressions are used to specify string patterns. You can use regular expressions whenever you need to locate strings that match a particular pattern. For example, one of our sample programs locates all hyperlinks in an HTML file by looking for strings of the pattern <a href=". . .">.

Of course, when specifying a pattern, the . . . notation is not precise enough. You need to specify exactly what sequence of

characters is a legal match, using a special syntax to describe a pattern.

In the following sections, we cover the regular expression syntax used by the Java API and discuss how to put regular expressions to work.

2.7.1. The Regular Expression Syntax

Let us start with a simple example. The regular expression

```
[Jj]ava.+
```

matches any string of the following form:

- The first letter is a J or j.
- The next three letters are ava.
- The remainder of the string consists of one or more arbitrary characters.

For example, the string "javanese" matches this particular regular expression, but the string "Core Java" does not.

As you can see, you need to know a bit of syntax to understand the meaning of a regular expression. Fortunately, for most purposes, a few straightforward constructs are sufficient.

In a regular expression, a character denotes itself unless it is one of the reserved characters

```
. * + ? { | ( ) [ \ ^ $
```

For example, the regular expression Java only matches the string Java.

The symbol . matches any single character. For example, .a.a matches Java and data.

The * symbol indicates that the preceding constructs may be repeated 0 or more times; for a +, it is 1 or more times. A suffix of ? indicates that a construct is optional (0 or 1 times). For example, be+s? matches be, bee, and bees. You can specify

other multiplicities with { }—see Table 2.13.

A | denotes an alternative: .(oo|ee)f matches beef or woof. Note the parentheses—without them, .oo|eef would be the alternative between .oo and eef. Parentheses are also used for grouping—see Section 2.7.4.

A character class is a set of character alternatives enclosed in brackets, such as [Jj], [0-9], [A-Za-z], or [^0-9]. Inside a character class, the - denotes a range (all characters whose Unicode values fall between the two bounds). However, a - that is the first or last character in a character class denotes itself. A ^ as the first character in a character class denotes the complement (all characters except those specified).

There are many predefined character classes such as \d (digits) or \s (spaces)— see Table 2.13. More complex character classes can be described with the \p syntax in Table 2.14. For example, \p{Sc} is the class of Unicode currency symbols.

The characters ^ and $ match the beginning and end of input.

If you need to have a literal . * + ? { | ( ) [\ ^ $, precede it by a backslash. Inside a character class, you only need to escape [and \, provided you are careful about the positions of] - ^. For example, []^-] is a class containing all three of them.

If the regular expression is in a string literal, each backslash needs to be escaped with another backslash. If you forget that second backslash, you usually get an error because sequences such as \$ or \. are not valid in string literals. But if you want to match a word boundary and accidentally use \b instead of \\b, then you have a problem: \b is a valid escape sequence, indicating a backspace.

Your IDE can help with that escaping. Paste the regular expression string into an empty string, and the backslashes will be doubled.

Instead of using backslashes, you can surround a string with \Q and \E. For example, \(\$0\.99\) and \Q($0.99)\E both match the string ($0.99).

If you have a string that may contain some of the many special characters in the regular expression syntax, you can escape them all by calling Parse.quote(str). This simply surrounds the string with \Q and \E, but it takes care of the special case where str may contain \E.

Expression	Description	Example
Characters		
c, not one of . * + ? { | ( ) [\ ^ $	The character *c*	J
.	Any character except line terminators, or any character if the DOTALL flag is set	
\X	Any Unicode “extended grapheme cluster”, which is perceived as a character or symbol	
\x{*p*}	The Unicode code point with hex code *p*	\x{1D546}
\u*hhhh*, \x*hh*, \0*o*, \0*oo*, \0*ooo*	The UTF-16 code unit with the given hex or octal value	\uFEFF
\a, \e, \f, \n, \r, \t	Alert (\x{7}), escape (\x{1B}), form feed (\x{B}), newline (\x{A}), carriage return	\n

	(\x{D}), tab (\x{9})	
\c*c*, where *c* is in [A-Z] or one of @ [\ ] ^ _ ?	The control character corresponding to the character *c*	\cH is a backspace (\x{8})
c, where *c* is not in [A-Za-z0-9]	The character *c*	\\
\Q. . .\E	Everything between the start and the end of the quotation	\Q(. . .)\E matches the string (. . .)
Character Classes		
[C_1C_2. . .], where C_i are characters, ranges *c*-*d*, or character classes	Any of the characters represented by C_1, C_2, . . .	[0-9+-]
[^. . .]	Complement of a character class	[^\d\s]
[. . .&&. . .]	Intersection of character classes	[\p{L}&&[^A-Za-z]]
\p{. . .}, \P{. . .}	A predefined character class (see Table 2.14); its complement	\p{L} matches a Unicode letter, and so does \pL—you can omit braces around a single letter
\d, \D	Digits ([0-9], or \p{Digit} when the UNICODE_CHARACTER_CLASS flag is set); the complement	\d+ is a sequence of digits
\w, \W	Word characters ([a-zA-Z0-9_], or Unicode word characters when the UNICODE_CHARACTER_CLASS flag is set); the complement	

\s, \S	Spaces ([\n\r\t\f\x{B}], or \p{IsWhite_Space} when the UNICODE_CHARACTER_CLASS flag is set); the complement	\s*,\s* is a comma surrounded by optional white space
\h, \v, \H, \V	Horizontal whitespace, vertical whitespace, their complements	
Sequences and Alternatives		
XY	Any string from X, followed by any string from Y	[1-9][0-9]* is a positive number without leading zero
X|*Y*	Any string from X or Y	http|ftp
Grouping		
(*X*)	Captures the match of *X*	'([^']*)' captures the quoted text ([""]).*\1 matches 'Fred' or "Fred" but not "Fred'
n	The *n*th group	
(?<*name*>*X*)	Captures the match of *X* with the given name	'(?<id>[A-Za-z0-9]+)' captures the match with name id
\k<*name*>	The group with the given name	\k<id> matches the group with name id
(?:*X*)	Use parentheses without capturing *X*	In (?:http|ftp)://(.*), the match after :// is \1
(?f_1f_2. . .:*X*), (?f_1. . .-f_k. . .:*X*), with f_i in [dimsuUx]	Matches, but does not capture, *X* with the given flags on or off	(?i:jpe?g) is a case-insensitive match

Other (?. . .)	(after -) See the Pattern API documentation	
Quantifiers		
X?	Optional *X*	\-? is an optional + sign
*X**, *X*+	0 or more *X*, 1 or more *X*	[1-9][0-9]+ is an integer ≥ 10
X{*n*}, *X*{*n*,}, *X*{*m*,*n*}	*n* times *X*, at least *n* times X, between *m* and *n* times *X*	[0-7]{1,3} are one to three octal digits
Q?, where *Q* is a quantified expression	Reluctant quantifier, attempting the shortest match before trying longer matches	.*(<.+?>).* captures the shortest sequence enclosed in angle brackets
Q+, where *Q* is a quantified expression	Possessive quantifier, taking the longest match without backtracking	'[^']*+' matches strings enclosed in single quotes and fails quickly on strings without a closing quote
Boundary Matches		
^ $	Beginning, end of input (or beginning, end of line in multiline mode)	^Java$ matches the input or line Java
\A \z \Z	Beginning of input, end of input, end of input before final line terminator	
\b \B	Word boundary, nonword boundary	\bJava\b matches the word Java

\b{g}	Grapheme cluster boundary	Useful with split to decompose a string into grapheme clusters
\R	A Unicode line break	
\G	The end of the previous match	

Table 2.13: Regular Expression Syntax

Name	Description
posixClass	*posixClass* is one of Lower, Upper, Alpha, Digit, Alnum, Punct, Graph, Print, Cntrl, XDigit, Space, Blank, ASCII, interpreted as POSIX or Unicode class, depending on the UNICODE_CHARACTER_CLASS flag
Is*Script*, sc=*Script*, script=*Script*	A script accepted by Character.UnicodeScript.forName
In*Block*, blk=*Block*, block=*Block*	A block accepted by Character.UnicodeBlock.forName
Category, In*Category*, gc=*Category*, general_category=*Category*	A one- or two-letter name for a Unicode general category
Is*Property*	*Property* is one of Alphabetic, Ideographic, Letter, Lowercase, Uppercase, Titlecase, Punctuation, Control, White_Space, Digit, Hex_Digit, Join_Control, Noncharacter_Code_Point, Assigned
java*Method*	Invokes the method Character.is*Method* (must

	not be deprecated)

Table 2.14: Predefined Character Classes \p{. . .}

Unfortunately, the regular expression syntax is not completely standardized between various programs and libraries; there is a consensus on the basic constructs but many maddening differences in the details. The Java regular expression classes use a syntax that is similar to, but not quite the same as, the one used in the Perl language. Table 2.13 shows all constructs of the Java syntax. For more information on the regular expression syntax, consult the API documentation for the Pattern class or the book Mastering Regular Expressions by Jeffrey E. F. Friedl (O'Reilly and Associates, 2006).

2.7.2. Testing a Match

Generally, there are two ways to use a regular expression: Either you want to find out whether a string conforms to the expression, or you want to find all matches of a regular expressions in a string.

In the first case, simply use the static matches method:

```
String regex = "-?\\d+";
CharSequence input = . . .;
if (Pattern.matches(regex, input)) {
   . . .
}
```

If you need to use the same regular expression many times, it is more efficient to compile it. Then, create a Matcher for each input:

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) . . .
```

If the match succeeds, you can retrieve the location of matched groups—see the following section.

If you want to test whether the input contains a match, use the find method instead:

```
if (matcher.find()) . . .
```

The find and match methods mutate the state of the Matcher object. If you just want to find out whether a given Matcher has already found a match, call the hasMatch method instead.

You can turn the pattern into a predicate:

```
Pattern digits = Pattern.compile("[0-9]+");
List<String> strings = List.of("December", "31st", "1999");
List<String> matchingStrings
   = strings.stream().filter(digits.asMatchPredicate()).toList(); // ["1999
```

The result contains all strings that match the regular expression.

Use the asPredicate method to test whether a string contains a match:

```
List<String> stringsContainingMatch
   = strings.stream().filter(digits.asPredicate()).toList(); // ["31st", "199
```

2.7.3. Finding All Matches in a String

In this section, we consider the other common use case for regular expressions—finding all matches in an input. Use this loop:

```
String input = . . .;
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
   String match = matcher.group();
   int matchStart = matcher.start();
   int matchEnd = matcher.end();
   . . .
}
```

In this way, you can process each match in turn. As shown in

the code fragment, you can get the matched string as well as its position in the input string.

More elegantly, you can call the results method to get a Stream<MatchResult>. The MatchResult interface has methods group, start, and end, just like Matcher. (In fact, the Matcher class implements this interface.) Here is how you get a list of all matches:

```
List<String> matches = pattern.matcher(input)
   .results()
   .map(MatchResult::group)
   .toList();
```

If you have the data in a file, then you can use the Scanner.findAll method to get a Stream<MatchResult>, without first having to read the contents into a string. You can pass a Pattern or a pattern string:

```
Scanner in = new Scanner(path, "UTF_8");
Stream<String> words = in.findAll("\\pL+")
   .map(MatchResult::group);
```

Listing 2.7 puts this mechanism to work. It locates all hypertext references in a web page and prints them. To run the program, supply a URL on the command line, such as

```
java v2ch02.regex.HrefMatch https://horstmann.com/index.html
```

Listing 2.7 **v2ch02/regex/HrefMatch.java**

```
package v2ch02.regex;

import module java.base;

/**
 * This program displays all hyperlinks in a web page by matching a regular ex
 * describes the <a href=...> HTML tag.
 */
class HrefMatch {
```

```
   void main(String[] args) throws Exception {
      String urlString = args.length > 0 ? args[0] : "https://openjdk.org/";

      // read contents of URL
      InputStream in = URI.create(urlString).toURL().openStream();
      var input = new String(in.readAllBytes());

      // search for all occurrences of pattern
      String patternString = "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>]*)\\s*>";
      Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITI
      pattern.matcher(input).results().map(MatchResult::group).forEach(IO::pri
   }
}
```

2.7.4. Groups

It is common to use groups for extracting components of a match. For example, suppose you have a line item in the invoice with item name, quantity, and unit price such as

```
   Blackwell Toaster    USD29.95
```

Here is a regular expression with groups for each component:

```
   (\p{Alnum}+(\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

After matching, you can extract the nth group from the matcher as

```
   String contents = matcher.group(n);
```

Groups are ordered by their opening parenthesis, starting at 1. (Group 0 is the entire input.) In this example, here is how to take the input apart:

```
   Matcher matcher = pattern.matcher(input);
   if (matcher.matches()) {
      item = matcher.group(1);
      currency = matcher.group(3);
      price = matcher.group(4);
   }
```

We aren't interested in group 2; it only arose from the parentheses that were required for the repetition. For greater clarity, you can use a noncapturing group:

```
(\p{Alnum}+(?:\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

Or, even better, use named groups:

```
(?<item>\p{Alnum}+(\s+\p{Alnum}+)*)\s+(?<currency>[A-Z]{3})(?<pri
```

Then, you can retrieve the items by name:

```
item = matcher.group("item");
```

With the start and end methods, you can get the group positions in the input:

```
int itemStart = matcher.start("item");
int itemEnd = matcher.end("item");
```

As of Java 20, the MatchResult class also supports named groups.

The namedGroups method yields a Map<String, Integer> from group names to numbers.

When you have a group inside a repetition, such as (\s+ \p{Alnum}+)* in the example above, it is not possible to get all of its matches. The group method only yields the last match, which is rarely useful. You need to capture the entire expression with another group.

You can add comments to regular expressions. A comment starts with # and extends until the end of the line. This works particularly well with text blocks:

```
var regex = """
([1-9]|1[0-2]) #hours
:([0-5][0-9]) #minutes
[ap]m""";
```

Listing 2.8 prompts for a pattern, then for strings to match. It prints out whether or not the input matches the pattern. If the input matches and the pattern contains groups, the program prints the group boundaries as parentheses, for example:

```
((11):(59))am
```

Listing 2.8 **v2ch02/regex/RegexDemo.java**

```
package v2ch02.regex;

/**
 * This program tests regular expression matching. Enter a pattern and strings
 * or hit Enter to exit. If the pattern contains groups, the group boundaries are
 * in the match.
 */

import module java.base;

class RegexDemo {
   void main() throws Exception {
      String patternString = IO.readln("Enter pattern: ");

      Pattern pattern = Pattern.compile(patternString);

      while (true) {
         String input = IO.readln("Enter string to match: ");
         if (input == null || input.equals("")) return;
         Matcher matcher = pattern.matcher(input);
         if (matcher.matches()) {
            IO.println("Match");
            int g = matcher.groupCount();
            if (g > 0) {
               for (int i = 0; i < input.length(); i++) {
                  // Print any empty groups
                  for (int j = 1; j <= g; j++)
                     if (i == matcher.start(j) && i == matcher.end(j)) IO.print("()");
                  // Print ( for non-empty groups starting here
                  for (int j = 1; j <= g; j++)
                     if (i == matcher.start(j) && i != matcher.end(j)) IO.print("(");
                  IO.print(input.charAt(i));
                  // Print ) for non-empty groups ending here
```

```
                for (int j = 1; j <= g; j++)
                   if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
                      IO.print(")");
             }
             IO.println();
          }
       }
       else
          IO.println("No match");
    }
  }
}
```

2.7.5. Splitting along Delimiters

Sometimes, you want to break an input along matched delimiters and keep everything else. The Pattern.split method automates this task. You obtain an array of strings, with the delimiters removed:

```
String input = "1, 2 , 3,4";
Pattern commas = Pattern.compile("\\s*,\\s*");
String[] tokens = commas.split(input); // ["1", "2", "3", "4"]
```

If there are many tokens, you can fetch them lazily:

```
Stream<String> tokens = commas.splitAsStream(input);
```

To also capture the delimiters, use the splitWithDelimiters method:

```
tokens = commas.splitWithDelimiters(input, -1); // ["1", ", ", "2", " , ", "
```

If the second argument is a positive number *n*, the separator pattern is applied at most *n* - 1 times, and the last element is the remaining string. Otherwise, the pattern is applied as many times as possible. With a limit of zero, trailing empty strings are discarded.

If you don't care about precompiling the pattern or lazy fetching, you can just use the split and splitWithDelimiter methods of the String class:

```
tokens = input.split("\\s*,\\s*");
```

It is easy to forget that the argument of split is a regular expression. For example,

```
"com.horstmann.corejava".split(".")
```

does not split along the dots. Instead, every character is a separator, and the result is an empty array!

You need to escape the dot with a backslash in the regular expression, and therefore with two backslashes in the string literal:

```
"com.horstmann.corejava".split("\\.")
```

Alternatively, use the Pattern.quote method:

```
"com.horstmann.corejava".split(Pattern.quote("."));
```

If the input is in a file, use a scanner:

```
Scanner in = new Scanner(path, "UTF_8");
in.useDelimiter("\\s*,\\s*");
Stream<String> tokens = in.tokens();
```

2.7.6. Replacing Matches

If you want to replace all matches of a regular expression with a string, call replaceAll on the matcher:

```
Matcher matcher = commas.matcher(input);
String result = matcher.replaceAll(",");
   // Normalizes the commas
```

Or, if you don't care about precompiling, use the replaceAll method of the String class.

```
String result = input.replaceAll("\\s*,\\s*", ",");
```

The replacement string can contain group numbers $*n* or

names ${*name*}. They are replaced with the contents of the corresponding captured groups.

```
String result = "3:45".replaceAll(
   "(\\d{1,2}):(?<minutes>\\d{2})",
   "$1 hours and ${minutes} minutes");
   // Sets result to "3 hours and 45 minutes"
```

You can use \ to escape $ and \ in the replacement string, or you can call the Matcher.quoteReplacement convenience method:

```
matcher.replaceAll(Matcher.quoteReplacement(str))
```

If you want to carry out a more complex operation than splicing in group matches, then you can provide a replacement function instead of a replacement string. The function accepts a MatchResult and yields a string. For example, here we replace all words with at least four letters with their uppercase version:

```
String result = Pattern.compile("\\pL{4,}")
   .matcher("Mary had a little lamb")
   .replaceAll(m -> m.group().toUpperCase());
   // Yields "MARY had a LITTLE LAMB"
```

The replaceFirst method replaces only the first occurrence of the pattern.

2.7.7. Flags

Several flags change the behavior of regular expressions. You can specify them when you compile the pattern:

```
Pattern pattern = Pattern.compile(regex,
   Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CHARACTER_CLAS
```

Or you can specify them inside the pattern:

```
String regex = "(?iU:expression)";
```

Here are the flags:

- Pattern.CASE_INSENSITIVE or i: Match characters independently of the letter case. By default, this flag takes only US ASCII characters into account.
- Pattern.UNICODE_CASE or u: When used in combination with CASE_INSENSITIVE, use Unicode letter case for matching.
- Pattern.UNICODE_CHARACTER_CLASS or U: Select Unicode character classes instead of POSIX. Implies UNICODE_CASE.
- Pattern.MULTILINE or m: Make ^ and $ match the beginning and end of a line, not the entire input.
- Pattern.UNIX_LINES or d: Only '\n' is a line terminator when matching ^ and $ in multiline mode.
- Pattern.DOTALL or s: Make the . symbol match all characters, including line terminators.
- Pattern.COMMENTS or x: Whitespace and comments (from # to the end of a line) are ignored.
- Pattern.LITERAL: The pattern is taken literally and must be matched exactly, except possibly for letter case.
- Pattern.CANON_EQ: Take canonical equivalence of Unicode characters into account. For example, u followed by ¨ (diaeresis) matches ü.

The last two flags cannot be specified inside a regular expression.

java.util.regex.Pattern1.4

- static Pattern compile(String expression)
- static Pattern compile(String expression, int flags)
  compile the regular expression string into a pattern object for fast processing of matches. The flags parameter has one or more of the bits CASE_INSENSITIVE, UNICODE_CASE, MULTILINE, UNIX_LINES, DOTALL, and CANON_EQ set.
- Matcher matcher(CharSequence input)
  returns a matcher object that you can use to locate the matches of the pattern in the input.
- String[] split(CharSequence input)

- String[] split(CharSequence input, int limit)
- String[] splitWithDelimiters(CharSequence input, int limit)
- Stream<String> splitAsStream(CharSequence input)8
  split the input string into tokens, with the pattern specifying the form of the delimiters. Return an array or stream of tokens. The delimiters are not part of the tokens. The second form has a parameter limit denoting the maximum number of strings to produce. If limit - 1 matching delimiters have been found, then the last entry of the returned array contains the remaining unsplit input. If limit is ≤0, then the entire input is split. If limit is 0, then trailing empty strings are not placed in the returned array.
- Map<String, Integer> namedGroups()20
  yields an immutable map from group names to group indexes.

java.util.regex.Matcher1.4

- boolean matches()
  returns true if the input matches the pattern.
- boolean lookingAt()
  returns true if the beginning of the input matches the pattern.
- boolean find()
- boolean find(int start)
  attempt to find the next match and return true if another match is found.
- boolean hasMatch()20
  true if this matcher is positioned on a match as a result of calling matches or find.
- String replaceAll(String replacement)
- String replaceFirst(String replacement)
  return a string obtained from the matcher input by replacing all matches, or the first match, with the replacement string. The replacement string can contain references to pattern groups as $*n*. Use \$ to include a $ symbol.
- static String quoteReplacement(String str)5.0

quotes all \ and $ in str.
- String replaceAll(Function<MatchResult,String> replacer)9
  replaces every match with the result of the replacer function applied to the MatchResult.
- Stream<MatchResult> results()9
  yields a stream of all match results.

java.util.regex.MatchResult5.0

- String group()
- String group(int group)
- String group(String name)20
  yield the matched string or the string matched by the given group.
- int start()
- int end()
- int start(int group)
- int start(String nane)20
- int end(int group)
- int end(String name)20
  yield the start and end offsets of the matched string or the string matched by the given group.
- Map<String, Integer> namedGroups()20
  yields an immutable map from group names to group indexes of the underlying pattern.

java.util.Scanner5.0

- Stream<MatchResult> findAll(Pattern pattern)9
  yields a stream of all matches of the given pattern in the input produced by this scanner.

You have now seen how to carry out input and output operations in Java, and had an overview of the regular expression package that was a part of the “new I/O” specification. In the next chapter, we turn to the processing of

XML data.

Chapter 3 ■ XML

The preface of the book Essential XML by Don Box et al. (Addison-Wesley, 2000) stated only half-jokingly: "The Extensible Markup Language (XML) has replaced Java, Design Patterns, and Object Technology as the software industry's solution to world hunger." This kind of hype is long gone but, as you will see in this chapter, XML is still a very useful technology for describing structured information. XML tools make it easy to process and transform that information. However, XML is not a silver bullet. You need domain-specific standards and code libraries to use it effectively. Moreover, far from making Java obsolete, XML works very well with Java. Since the late 1990s, IBM, Apache, and others have been instrumental in producing high-quality Java libraries for XML processing. Many of these libraries have now been integrated into the Java platform.

This chapter introduces XML and covers the XML features of the Java API. As always, we'll point out, along the way, when using XML is justified—and when you have to take it with a grain of salt and try solving your problems the old-fashioned way: through good design and code.

3.1. Introducing XML

In Chapter 9 of Volume I, you have seen the use of property files to describe the configuration of a program. A property file contains a set of name/value pairs, such as

```
fontname=Times Roman
fontsize=12
windowsize=400 200
color=0 50 100
```

You can use the Properties class to read in such a file with a single method call. That's a nice feature, but it doesn't really go far enough. In many cases, the information you want to describe has more structure than the property file format can comfortably handle. Consider the fontname/fontsize entries in the example. It would be more object-oriented to have a single entry:

```
font=Times Roman 12
```

But then, parsing the font description gets ugly as you have to figure out when the font name ends and the font size starts.

Property files have a single flat hierarchy. You can often see programmers work around that limitation with key names like

```
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12
```

Another shortcoming of the property file format is the requirement that keys must be unique. To store a sequence of values, you need another workaround, such as

```
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
```

The XML format solves these problems. It can express hierarchical structures and is thus more flexible than the flat table structure of a property file.

An XML file for describing a program configuration might look like this:

```
<config>
```

```
    <entry id="title">
      <font>
        <name>Helvetica</name>
        <size>36</size>
      </font>
    </entry>
    <entry id="body">
      <font>
        <name>Times Roman</name>
        <size>12</size>
      </font>
    </entry>
    <entry id="background">
      <color>
        <red>0</red>
        <green>50</green>
        <blue>100</blue>
      </color>
    </entry>
  </config>
```

The XML format allows you to express the hierarchy and record repeated elements without contortions.

The format of an XML file is straightforward. It looks similar to an HTML file. There is a good reason for that—both XML and HTML are descendants of the venerable Standard Generalized Markup Language (SGML).

SGML has been around since the 1970s for describing the structure of complex documents. It has been used with success in some industries that require ongoing maintenance of massive documentation—in particular, the aircraft industry. However, SGML is quite complex, so it has never caught on in a big way. Much of that complexity arises because SGML has two conflicting goals. SGML wants to make sure that documents are formed according to the rules for their document type, but it also wants to make data entry easy by allowing shortcuts that reduce typing. XML was designed as a simplified version of SGML for use on the Internet. As is often true, simpler is better,

and XML has enjoyed the immediate and enthusiastic reception that has eluded SGML for so long.

> You can find a very nice version of the XML standard, with annotations by Tim Bray, at https://www.xml.com/axml/axml.html.

Even though XML and HTML have common roots, there are important differences between the two.

- Unlike HTML, XML is case-sensitive. For example, <H1> and <h1> are different XML tags.
- In HTML, you can omit end tags, such as </p> or , if it is clear from the context where a paragraph or list item ends. In XML, you can never omit an end tag.
- In XML, elements that have a single tag without a matching end tag must end in a /, as in . That way, the parser knows not to look for a </img> tag.
- In XML, attribute values must be enclosed in quotation marks. In HTML, quotation marks are optional. For example, <img src="my hamster.png" width=300> is legal HTML but not legal XML. In XML, you have to use quotation marks: width="300".
- In HTML, you can have attribute names without values, such as <input type="radio" name="language" value="Java" checked>. In XML, all attributes must have values, such as checked="true" or (ugh) checked="checked".
- There are XML formulations for HTML versions 4 and 5 that are known as XHTML.

3.2. The Structure of an XML Document

An XML document should start with a header such as

```
<?xml version="1.0"?>
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
```

Strictly speaking, a header is optional, but it is highly recommended.

> Since SGML was created for processing of real documents, XML files are called documents even though many of them describe data sets that one would not normally call documents.

The header can be followed by a document type definition (DTD), such as

```
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

DTDs are an important mechanism to ensure the correctness of a document, but they are not required. We will discuss them later in this chapter.

Finally, the body of the XML document contains the root element, which can contain other elements. For example,

```
<?xml version="1.0"?>
<!DOCTYPE config . . .>
<config>
  <entry id="title">
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </entry>
  . . .
</config>
```

An element can contain child elements, text, or both. In the preceding example, the font element has two child elements, name and size. The name element contains the text "Helvetica".

It is best to structure your XML documents so that an element contains either child elements or text. Here's an example of what you should avoid:

```
<font>
  Helvetica
  <size>36</size>
</font>
```

This is called mixed content in the XML specification. As you will see later in this chapter, you can simplify parsing if you avoid mixed content.

XML elements can contain attributes, such as

```
<size unit="pt">36</size>
```

There is some disagreement among XML designers about when to use elements and when to use attributes. For example, it would seem easier to describe a font as

```
<font name="Helvetica" size="36"/>
```

compared to

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

However, attributes are much less flexible. Suppose you want to add units to the size value. If you use attributes, you will have to add the unit to the attribute value:

```
<font name="Helvetica" size="36 pt"/>
```

Ugh! Now you have to parse the string "36 pt", just the kind of hassle that XML was designed to avoid. Adding an attribute to the size element is much cleaner:

```
<font>
  <name>Helvetica</name>
```

```
    <size unit="pt">36</size>
  </font>
```

A commonly used rule of thumb is that attributes should be used only to modify the interpretation of a value, not to specify values. If you find yourself engaged in a metaphysical discussion about whether a particular setting is a modification of the interpretation of a value or not, just say "no" to attributes and use elements throughout. Many useful XML documents don't use attributes at all.

> In HTML, the rule for attribute usage is simple: If it isn't displayed on the web page, it's an attribute. For example, consider the hyperlink
>
> ```
> <a href="http://java.sun.com">Java Technology</a>
> ```
>
> The string Java Technology is displayed on the web page, but the URL of the link is not a part of the displayed page. However, the rule isn't all that helpful for most XML files because the data in an XML file aren't normally meant to be viewed by humans.

Elements and text are the "bread and butter" of XML documents. Here are a few other markup instructions that you might encounter:

- Character references have the form &#decimalValue; or &#xhexValue;. For example, the é character can be denoted with either of the following:

  ```
  &#233; &#xE9;
  ```

- Entity references have the form &name;. The entity references

  ```
  &lt; &gt; &amp; &quot; &apos;
  ```

  have predefined meanings: the less-than, greater-than, ampersand, quotation mark, and apostrophe characters.

You can define other entity references in a DTD.

- CDATA sections are delimited by <![CDATA[and]]>. They are a special form of character data. You can use them to include strings that contain characters such as < > & without having them interpreted as markup, for example:

  ```
  <![CDATA[< & > are my favorite delimiters]]>
  ```

  CDATA sections cannot contain the string]]>. Use this feature with caution! It is too often used as a back door for smuggling legacy data into XML documents.
- Processing instructions are instructions for applications that process XML documents. They are delimited by <? and ?>, for example

  ```
  <?xml-stylesheet href="mystyle.css" type="text/css"?>
  ```

  Every XML document starts with a processing instruction

  ```
  <?xml version="1.0"?>
  ```

- Comments are delimited by <!-- and -->, for example

  ```
  <!-- This is a comment. -->
  ```

  Comments should not contain the string --. Comments should only be information for human readers. They should never contain hidden commands; use processing instructions for commands.

3.3. Parsing an XML Document

To process an XML document, you need to parse it. A parser is a program that reads a file, confirms that the file has the correct format, breaks it up into the constituent elements, and lets a programmer access those elements. The Java API supplies two kinds of XML parsers:

- Tree parsers, such as a Document Object Model (DOM) parser, that read an XML document into a tree structure.
- Streaming parsers, such as a Simple API for XML (SAX)

parser, that generate events as they read an XML document.

DOM parsers are easier to use for most purposes, so we explain them first. You may consider a streaming parser if you process very long documents whose tree structures would use up a lot of memory, or if you are only interested in a few elements and don't care about their context. For more information, see Section 3.7.

The DOM parser interface is standardized by the World Wide Web Consortium (W3C). The org.w3c.dom package contains the definitions of interface types such as Document and Element. Different suppliers, such as the Apache Organization and IBM, have written DOM parsers whose classes implement these interfaces. The Java API for XML Processing (JAXP) actually makes it possible to plug in any of these parsers. But the JDK also comes with a DOM parser that is derived from the Apache parser.

To read an XML document, you need a DocumentBuilder object that you get from a DocumentBuilderFactory like this:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newIns
DocumentBuilder builder = factory.newDocumentBuilder();
```

You can now read a document from a file:

```
File f = . . .;
Document doc = builder.parse(f);
```

Alternatively, you can use a URL string:

```
String urlString = . . .;
Document doc = builder.parse(urlString);
```

You can even specify an arbitrary input stream:

```
InputStream in = . . .;
Document doc = builder.parse(in);
```

If you use an input stream as an input source, the parser will not be able to locate other files that are referenced relative to the location of the document, such as a DTD in the same directory. You can install an "entity resolver" to overcome that problem. See https://www.xml.com/pub/a/2004/03/03/catalogs.html and https://openjdk.org/jeps/268 for more information.

A Document object is an in-memory representation of the tree structure of an XML document. It is composed of objects whose classes implement the Node interface and its various subinterfaces. Figure 3.1 shows the inheritance hierarchy of the subinterfaces.

.

Figure 3.1: The Node interface and its subinterfaces

Start analyzing the contents of a document by calling the getDocumentElement method. It returns the root element.

```
Element root = doc.getDocumentElement();
```

For example, if you are processing a document

```
<?xml version="1.0"?>
<font>
  . . .
</font>
```

then calling getDocumentElement returns the font element.

The getTagName method returns the tag name of an element. In the preceding example, root.getTagName() returns the string "font".

To get an element's children (which may be subelements, text, comments, or other nodes), use the getChildNodes method. That method returns a collection of type NodeList. That type was standardized before the standard Java collections, so it has

a different access protocol. The item method gets the item with a given index, and the getLength method gives the total count of the items. You can enumerate all children like this:

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++) {
   Node child = children.item(i);
   . . .
}
```

Be careful when analyzing children. Suppose, for example, that you are processing the document

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

You would expect the font element to have two children, but the parser reports five:

- The whitespace between and <name>
- The name element
- The whitespace between </name> and <size>
- The size element
- The whitespace between </size> and

Figure 3.2 shows the DOM tree.

.

Figure 3.2: A simple DOM tree

If you expect only subelements, you can ignore the whitespace:

```
for (int i = 0; i < children.getLength(); i++) {
   Node child = children.item(i);
   if (child instanceof Element childElement) {
      . . .
   }
```

```
}
```

Now you look at only two elements, with tag names name and size.

As you will see in the next section, you can do even better if your document has a DTD. Then the parser knows which elements don't have text nodes as children, and it can suppress the whitespace for you.

When analyzing the name and size elements, you want to retrieve the text strings that they contain. Those text strings are themselves contained in child nodes of type Text. You know that these Text nodes are the only children, so you can use the getFirstChild method without having to traverse another NodeList. Then, use the getData method to retrieve the string stored in a Text node:

```
for (int i = 0; i < children.getLength(); i++) {
   Node child = children.item(i);
   if (child instanceof Element childElement) {
      var textNode = (Text) childElement.getFirstChild();
      String text = textNode.getData().strip();
      if (childElement.getTagName().equals("name"))
         name = text;
      else if (childElement.getTagName().equals("size"))
         size = Integer.parseInt(text);
   }
}
```

It is a good idea to call strip on the return value of the getData method. If the author of an XML file puts the beginning and the ending tags on separate lines, such as

```
<size>
  36
</size>
```

then the parser will include all line breaks and spaces in the text node data. Calling the strip method removes the

---

whitespace surrounding the actual data.

You can also get the last child with the getLastChild method, and the next sibling of a node with getNextSibling. Therefore, another way of traversing a node's children is

```
for (Node childNode = element.getFirstChild();
      childNode != null;
      childNode = childNode.getNextSibling()) {
   . . .
}
```

To enumerate the attributes of a node, call the getAttributes method. It returns a NamedNodeMap object that contains Node objects describing the attributes. You can traverse the nodes in a NamedNodeMap the same way as in a NodeList. Then, call the getNodeName and getNodeValue methods to get the attribute names and values.

```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++) {
   Node attribute = attributes.item(i);
   String name = attribute.getNodeName();
   String value = attribute.getNodeValue();
   . . .
}
```

Alternatively, if you know the name of an attribute, you can retrieve the corresponding value directly:

```
String unit = element.getAttribute("unit");
```

You have now seen how to analyze a DOM tree. The program in Listing 3.1 puts these techniques to work by converting an XML document to JSON format.

The tree display clearly shows how child elements are surrounded by text containing whitespace and comments. You can clearly see the newline and return characters as \n.

You don't have to be familiar with JSON to understand how the

program works with the DOM tree. Simply observe the following:

- We use a DocumentBuilder to read a Document from a file.
- For each element, we print the tag name, attributes, and elements.
- For character data, we produce a string with the data. If the data comes from a comment, we add a "Comment: " prefix.

Listing 3.1 **v2ch03/dom/JSONConverter.java**

```
package v2ch03.dom;

import module java.base;
import module java.xml;
import org.w3c.dom.CharacterData;
import org.w3c.dom.Comment;

/**
 * This program displays an XML document as a tree in JSON format.
 */
class JSONConverter {
   void main(String[] args) throws Exception {
      String filename = args.length == 0 ? IO.readln("Input file: ") : args[0];
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance
      DocumentBuilder builder = factory.newDocumentBuilder();

      Document doc = builder.parse(filename);
      Element root = doc.getDocumentElement();
      IO.println(convert(root, 0));
   }

   StringBuilder convert(Node node, int level) {
      if (node instanceof Element elem) {
         return elementObject(elem, level);
      }
      else if (node instanceof CharacterData cd) {
         return characterString(cd, level);
      }
```

```
    else {
        return pad(new StringBuilder(), level)
                .append(jsonEscape(node.getClass().getName()));
    }
}

Map<Character, String> replacements = Map.of('\b', "\\b", '\f', "\\f", '\n',
        "\\n", '\r', "\\r", '\t', "\\t", '"', "\\\"", '\\', "\\\\");

StringBuilder jsonEscape(String str) {
    var result = new StringBuilder("\"");
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        String replacement = replacements.get(ch);
        if (replacement == null)
            result.append(ch);
        else
            result.append(replacement);
    }
    result.append("\"");
    return result;
}

StringBuilder characterString(CharacterData node, int level) {
    var result = new StringBuilder();
    StringBuilder data = jsonEscape(node.getData());
    if (node instanceof Comment) data.insert(1, "Comment: ");
    pad(result, level).append(data);
    return result;
}

StringBuilder elementObject(Element elem, int level) {
    var result = new StringBuilder();
    pad(result, level).append("{\n");
    pad(result, level + 1).append("\"name\": ");
    result.append(jsonEscape(elem.getTagName()));
    NamedNodeMap attrs = elem.getAttributes();
    if (attrs.getLength() > 0) {
        pad(result.append(",\n"), level + 1).append("\"attributes\": ");
        result.append(attributeObject(attrs));
    }
    NodeList children = elem.getChildNodes();
    if (children.getLength() > 0) {
        pad(result.append(",\n"), level + 1).append("\"children\": [\n");
```

```
            for (int i = 0; i < children.getLength(); i++) {
               if (i > 0) result.append(",\n");
               result.append(convert(children.item(i), level + 2));
            }
            result.append("\n");
            pad(result, level + 1).append("]\n");
         }
         pad(result, level).append("}");
         return result;
      }

      StringBuilder pad(StringBuilder builder, int level) {
         for (int i = 0; i < level; i++)
            builder.append("  ");
         return builder;
      }

      StringBuilder attributeObject(NamedNodeMap attrs) {
         var result = new StringBuilder("{");
         for (int i = 0; i < attrs.getLength(); i++) {
            if (i > 0) result.append(", ");
            result.append(jsonEscape(attrs.item(i).getNodeName()));
            result.append(": ");
            result.append(jsonEscape(attrs.item(i).getNodeValue()));
         }
         result.append("}");
         return result;
      }
   }
```

javax.xml.parsers.DocumentBuilderFactory 1.4

- static DocumentBuilderFactory newInstance()
  returns an instance of the DocumentBuilderFactory class.
- DocumentBuilder newDocumentBuilder()
  returns an instance of the DocumentBuilder class.

javax.xml.parsers.DocumentBuilder 1.4

- Document parse(File f)
- Document parse(String url)
- Document parse(InputStream in)

parse an XML document from the given file, URL, or input stream and return the parsed document.

org.w3c.dom.Document1.4

- Element getDocumentElement()
  returns the root element of the document.

org.w3c.dom.Element1.4

- String getTagName()
  returns the name of the element.
- String getAttribute(String name)
  returns the value of the attribute with the given name, or an empty string if there is no such attribute.

org.w3c.dom.Node1.4

- NodeList getChildNodes()
  returns a node list that contains all children of this node.
- Node getFirstChild()
- Node getLastChild()
  get the first or last child node of this node, or null if this node has no children.
- Node getNextSibling()
- Node getPreviousSibling()
  get the next or previous sibling of this node, or null if this node has no siblings.
- Node getParentNode()
  gets the parent of this node, or null if this node is the document node.
- NamedNodeMap getAttributes()
  returns a node map that contains Attr nodes that describe all attributes of this node.
- String getNodeName()
  returns the name of this node. If the node is an Attr node,

the name is the attribute name.
- String getNodeValue()
  returns the value of this node. If the node is an Attr node, the value is the attribute value.

org.w3c.dom.CharacterData1.4

- String getData()
  returns the text stored in this node.

org.w3c.dom.NodeList1.4

- int getLength()
  returns the number of nodes in this list.
- Node item(int index)
  returns the node with the given index. The index is between 0 and getLength()– 1.

org.w3c.dom.NamedNodeMap1.4

- int getLength()
  returns the number of nodes in this map.
- Node item(int index)
  returns the node with the given index. The index is between 0 and getLength()– 1.

3.4. Validating XML Documents

In the previous section, you saw how to traverse the tree structure of a DOM document. However, with that approach, you'll have to do quite a bit of tedious programming and error checking. It's not just having to deal with whitespace between elements; you will also need to check whether the document contains the nodes that you expect. For example, suppose you

are reading this element:

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

You get the first child. Oops . . . it is a text node containing whitespace "\n ". You skip text nodes and find the first element node. Then, you need to check that its tag name is "name" and that it has one child node of type Text. You move on to the next nonwhitespace child and make the same check. What if the author of the document switched the order of the children or added another child element? It is tedious to code all this error checking—but reckless to skip the checks.

Fortunately, one of the major benefits of an XML parser is that it can automatically verify that a document has the correct structure. That makes parsing much simpler. For example, if you know that the font fragment has passed validation, you can simply get the two grandchildren, cast them as Text nodes, and get the text data, without any further checking.

To specify the document structure, you can supply a DTD or an XML Schema definition. A DTD or schema contains rules that explain how a document should be formed, by specifying the legal child elements and attributes for each element. For example, a DTD might contain a rule:

```
<!ELEMENT font (name,size)>
```

This rule expresses that a font element must always have two children, which are name and size elements. The XML Schema language expresses the same constraint as

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

XML Schema can express more sophisticated validation conditions (such as the fact that the size element must contain an integer) than DTDs. Unlike the DTD syntax, the XML Schema syntax itself uses XML, which is a benefit if you need to process schema files.

In the next section, we will discuss DTDs in detail, then briefly cover the basics of XML Schema support. Finally, we will present a complete application that demonstrates how validation simplifies XML programming.

3.4.1. Document Type Definitions

There are several methods for supplying a DTD. You can include a DTD in an XML document like this:

```
<?xml version="1.0"?>
<!DOCTYPE config [
  <!ELEMENT config . . .>
  more rules
  . . .
]>
<config>
  . . .
</config>
```

As you can see, the rules are included inside a DOCTYPE declaration, in a block delimited by [. . .]. The document type must match the name of the root element, such as config in our example.

Supplying a DTD inside an XML document is somewhat uncommon because DTDs can grow lengthy. It makes more sense to store the DTD externally. The SYSTEM declaration can be used for that purpose. Specify a URL that contains the DTD, for example:

```
<!DOCTYPE config SYSTEM "config.dtd">
```

or

```
<!DOCTYPE config SYSTEM "http://myserver.com/config.dtd">
```

If you use a relative URL for the DTD (such as "config.dtd"), give the parser a File or URL object, not an InputStream. If you must parse from an input stream, supply an entity resolver (see the following note).

The mechanism for identifying well-known DTDs has its origin in SGML. Here is an example:

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

If an XML processor knows how to locate the DTD with the public identifier, it need not go to the URL.

The system identifier URL of a DTD may not actually be working, or be purposefully slowed down. An example for the latter is the system identifier of the XHTML 1.0 Strict DTD, http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd. If you parse an XHTML file, it may take a minute or two for the DTD to be served.

The remedy is to use an entity resolver that maps public identifiers to local files. You provide an object of a class that implements the EntityResolver interface and provides a resolveEntity method.

XML catalogs are a convenient mechanism for resolving entities. Provide one or more catalog files of the form

```
<?xml version="1.0"?>
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.0/
    "http://www.oasis-open.org/committees/entity/release/1.0/c
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
  <public publicId=". . ." uri=". . ."/>
  . . .
</catalog>
```

Then construct and install a resolver like this:

```
builder.setEntityResolver(CatalogManager.catalogResolver(
  CatalogFeatures.defaults(),
  Path.of("catalog.xml").toAbsolutePath().toUri()));
```

See Listing 3.6 for a complete example.

Instead of setting the catalog file locations in your program, you can specify them on the command line with the javax.xml.catalog.files system property. Provide absolute file URLs separated by semicolons.

Now that you have seen how the parser locates the DTD, let us consider the various rules.

The ELEMENT rule specifies what children an element can have. Use a regular expression, made up of the components shown in #ch02table01.

Meaning
E^* 0 or more occurrences of E
E^+ 1 or more occurrences of E
$E?$ 0 or 1 occurrences of E
$E_1 \mid E_2 \mid \ldots \mid E_n$ One of $E_1, E_2, \ldots, E_n$
$E_1, E_2, \ldots, E_n$ E_1 followed by $E_2, \ldots, E_n$
#PCDATA Text
(#PCDATA $\mid E_1 \mid E_2 \mid \ldots \mid E_n$)* Text and $E_1, E_2, \ldots, E_n$ in any order (mixed content)
ANY Any children allowed
EMPTY No children allowed

Table 3.1: Rules for Element Content

Here are several simple but typical examples. The following rule states that a menu element contains 0 or more item elements:

```
<!ELEMENT menu (item)*>
```

This set of rules states that a font is described by a name

followed by a size, each of which contain text:

```
<!ELEMENT font (name,size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

The abbreviation PCDATA denotes parsed character data. It is "parsed" because the parser interprets the text string, looking for < characters that denote the start of a new tag, or & characters that denote the start of an entity.

An element specification can contain regular expressions that are nested and complex. For example, here is a rule that describes the makeup of a chapter in a book:

```
<!ELEMENT chapter (intro,(heading,(para|image|table|note)+)+)
```

Each chapter starts with an introduction, which is followed by one or more sections consisting of a heading and one or more paragraphs, images, tables, or notes.

However, in one common case you can't define the rules to be as flexible as you might like. Whenever an element can contain text, there are only two valid cases. Either the element contains nothing but text, such as

```
<!ELEMENT name (#PCDATA)>
```

or the element contains any combination of text and tags in any order, such as

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

It is not legal to specify any other types of rules that contain #PCDATA. For example, the following is illegal:

```
<!ELEMENT captionedImage (image,#PCDATA)>
```

You have to rewrite such a rule, either by introducing another caption element or by allowing any combination of image elements and text.

This restriction simplifies the job of the XML parser when parsing mixed content (a mixture of tags and text). Since you lose some control by allowing mixed content, it is best to design DTDs so that all elements contain either other elements or nothing but text.

Actually, it isn't quite true that you can specify arbitrary regular expressions of elements in a DTD rule. An XML parser may reject certain complex rule sets that lead to nondeterministic parsing. For example, a regular expression ((x,y)|(x,z)) is nondeterministic. When the parser sees x, it doesn't know which of the two alternatives to take. This expression can be rewritten in a deterministic form as (x,(y|z)). However, some expressions can't be reformulated, such as ((x,y)*|x?). The parser in the Java XML API gives no warnings when presented with an ambiguous DTD; it simply picks the first matching alternative when parsing, which causes it to reject some correct inputs. The parser is well within its rights to do so because the XML standard allows a parser to assume that the DTD is unambiguous.

You can also specify rules to describe the legal attributes of elements. The general syntax is

```
<!ATTLIST element attribute type default>
```

#ch02table02 shows the legal attribute types, and #ch02table03 shows the syntax for the defaults.

Type	Meaning
CDATA	Character string
(A_1|A_2|...)	One of the strings A_1, A_2, ..., A_n
NMTOKEN, NMTOKENS	Name tokens
ID	Unique ID
IDREF, IDREFS	References to a unique ID
ENTITY, ENTITIES	Unparsed entities

Table 3.2: Attribute Types

Default	Meaning
#REQUIRED	Required.
#IMPLIED	Is optional.
A	Attribute is optional; the parser reports it to be A if it is not specified.
#FIXED A	The attribute must either be unspecified or A; in either case, the parser reports it to be A.

Table 3.3: Attribute Defaults

Here are two typical attribute specifications:

```
<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">
<!ATTLIST size unit CDATA #IMPLIED>
```

The first specification describes the style attribute of a font element. There are four legal attribute values, and the default value is plain. The second specification expresses that the unit attribute of the size element can contain any character data sequence.

> It is generally recommended to use elements, not attributes, to describe data. Thus, the font style should be a separate element, such as <style>plain</style>. . .. However, attributes have an undeniable advantage for enumerated types because the parser can verify that the values are legal. For example, if the font style is an attribute, the parser checks that it is one of the four allowed values, and supplies a default if no value was given.

The handling of a CDATA attribute value is subtly different from the processing of #PCDATA that you have seen before, and quite unrelated to the <![CDATA[. . .]]> sections. The attribute value is first normalized—that is, the parser processes character and entity references (such as é or <) and replaces whitespace with spaces.

An NMTOKEN (or name token) is similar to CDATA, but most

nonalphanumeric characters and internal whitespace are disallowed, and the parser removes leading and trailing whitespace. NMTOKENS is a whitespace-separated list of name tokens.

The ID construct is quite useful. An ID is a name token that must be unique in the document—the parser checks the uniqueness. You will see an application in the next sample program. An IDREF is a reference to an ID that exists in the same document, which the parser also checks. IDREFS is a whitespace-separated list of ID references.

An ENTITY attribute value refers to an "unparsed external entity." That is a holdover from SGML that is rarely used in practice. The annotated XML specification at https://www.xml.com/axml/axml.html has an example.

A DTD can also define entities, or abbreviations that are replaced during parsing. You can find a good example for the use of entities in the user interface descriptions of the Firefox browser. Those descriptions are formatted in XML and contain entity definitions such as

```
<!ENTITY back.label "Back">
```

Elsewhere, text can contain an entity reference, for example:

```
<menuitem label="&back.label;"/>
```

The parser replaces the entity reference with the replacement string. To internationalize the application, only the string in the entity definition needs to be changed. Other uses of entities are more complex and less common; look at the XML specification for details.

This concludes the introduction to DTDs. Now that you have seen how to use DTDs, you can configure your parser to take advantage of them.

First, tell the document builder factory to turn on validation:

```
factory.setValidating(true);
```

All builders produced by this factory validate their input against a DTD. The most useful benefit of validation is ignoring whitespace in element content. For example, consider the XML fragment

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

A nonvalidating parser reports the whitespace between the font, name, and size elements because it has no way of knowing if the children of font are

```
(name,size)
(#PCDATA,name,size)*
```

or perhaps

```
ANY
```

Once the DTD specifies that the children are (name,size), the parser knows that the whitespace between them is not text. Call

```
factory.setIgnoringElementContentWhitespace(true);
```

and the builder will stop reporting the whitespace in text nodes. That means you can now rely on the fact that a font node has two children. You no longer need to program a tedious loop:

```
for (int i = 0; i < children.getLength(); i++) {
   Node child = children.item(i);
   if (child instanceof Element childElement) {
      if (childElement.getTagName().equals("name")) . . .;
      else if (childElement.getTagName().equals("size")) . . .;
   }
}
```

Instead, you can simply access the first and second child:

```
var nameElement = (Element) children.item(0);
```

```
var sizeElement = (Element) children.item(1);
```

That is why DTDs are so useful. You don't overload your program with rule-checking code—the parser has already done that work by the time you get the document.

When the parser reports an error, your application will want to do something about it—log it, show it to the user, or throw an exception to abandon the parsing. Therefore, you should install an error handler whenever you use validation. Supply an object that implements the ErrorHandler interface. That interface has three methods:

```
void warning(SAXParseException e)
void error(SAXParseException e)
void fatalError(SAXParseException e)
```

Install the error handler with the setErrorHandler method of the DocumentBuilder class:

```
builder.setErrorHandler(handler);
```

javax.xml.parsers.DocumentBuilder 1.4

- void setEntityResolver(EntityResolver resolver)
  sets the resolver to locate entities that are referenced in the XML documents being parsed.
- void setErrorHandler(ErrorHandler handler)
  sets the handler to report errors and warnings that occur during parsing.

org.xml.sax.EntityResolver 1.4

- public InputSource resolveEntity(String publicID, String systemID)
  returns an input source that contains the data referenced by the given ID(s), or null to indicate that this resolver doesn't know how to resolve the particular name. The publicID parameter may be null if no public ID was

supplied.

org.xml.sax.InputSource1.4

- InputSource(InputStream in)
- InputSource(Reader in)
- InputSource(String systemID)
  construct an input source from a stream, reader, or system ID (usually a relative or absolute URL).

org.xml.sax.ErrorHandler1.4

- void fatalError(SAXParseException e)
- void error(SAXParseException e)
- void warning(SAXParseException e)
  Override these methods to provide handlers for fatal errors, nonfatal errors, and warnings.

org.xml.sax.SAXParseException1.4

- int getLineNumber()
- int getColumnNumber()
  return the line and column numbers of the end of the processed input that caused the exception.

javax.xml.catalog.CatalogManager9

- static CatalogResolver catalogResolver(CatalogFeatures features, URI... uris)
  produces a resolver that uses the catalog files located at the provided URIs. This class implements EntityResolver as well as the resolver classes used by StAX, schema validation, and XSL transforms.

javax.xml.catalog.CatalogFeatures9

- static CatalogFeatures defaults()
  yields an instance with default settings.

javax.xml.parsers.DocumentBuilderFactory1.4

- boolean isValidating()
- void setValidating(boolean value)
  get or set the validating property of the factory. If set to true, the parsers that this factory generates validate their input.
- boolean isIgnoringElementContentWhitespace()
- void setIgnoringElementContentWhitespace(boolean value)
  get or set the ignoringElementContentWhitespace property of the factory. If set to true, the parsers that this factory generates ignore whitespace between element nodes that don't have mixed content (i.e., a mixture of elements and #PCDATA).

3.4.2. XML Schema

XML Schema is quite a bit more complex than the DTD syntax, so we will only cover the basics. For more information, we recommend the tutorial at https://www.w3.org/TR/xmlschema-0.

To reference a schema file in a document, add attributes to the root element, for example:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
  . . .
</config>
```

This declaration states that the schema file config.xsd should be used to validate the document. If your document uses namespaces, the syntax is a bit more complex—see the XML Schema tutorial for details. (The prefix xsi is a namespace alias; see Section 3.6 for more information.)

A schema defines a type for each element and attribute. A simple type is a string, perhaps with restrictions on its contents. Everything else is a complex type. An element with a simple type can have no attributes and no child elements. Otherwise, it must have a complex type. Conversely, attributes always have a simple type.

Some simple types are built into XML Schema, including

```
xsd:string
xsd:int
xsd:boolean
```

> We use the prefix xsd: to denote the XML Schema Definition namespace. Some authors use the prefix xs: instead.

You can define your own simple types. For example, here is an enumerated type:

```
<xsd:simpleType name="StyleType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="PLAIN" />
    <xsd:enumeration value="BOLD" />
    <xsd:enumeration value="ITALIC" />
    <xsd:enumeration value="BOLD_ITALIC" />
  </xsd:restriction>
</xsd:simpleType>
```

When you define an element, you specify its type:

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="size" type="xsd:int"/>
<xsd:element name="style" type="StyleType"/>
```

The type constrains the element content. For example, the elements

```
<size>10</size>
<style>PLAIN</style>
```

will validate correctly, but the elements

```
<size>default</size>
<style>SLANTED</style>
```

will be rejected by the parser.

You can compose types into complex types, for example:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element ref="name"/>
    <xsd:element ref="size"/>
    <xsd:element ref="style"/>
  </xsd:sequence>
</xsd:complexType>
```

A FontType is a sequence of name, size, and style elements. In this type definition, we use the ref attribute and refer to definitions that are located elsewhere in the schema. You can also nest definitions, like this:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
```

```
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
```

Note the anonymous type definition of the style element.

The xsd:sequence construct is the equivalent of the concatenation notation in DTDs. The xsd:choice construct is the equivalent of the | operator. For example,

```
  <xsd:complexType name="contactinfo">
    <xsd:choice>
      <xsd:element ref="email"/>
      <xsd:element ref="phone"/>
    </xsd:choice>
  </xsd:complexType>
```

This is the equivalent of the DTD type email|phone.

To allow repeated elements, use the minoccurs and maxoccurs attributes. For example, the equivalent of the DTD type item* is

```
  <xsd:element name="item" type=". . ." minoccurs="0" maxoccurs="ur
```

To specify attributes, add xsd:attribute elements to complexType definitions:

```
  <xsd:element name="size">
    <xsd:complexType>
      . . .
      <xsd:attribute name="unit" type="xsd:string" use="optional" defa
    </xsd:complexType>
  </xsd:element>
```

This is the equivalent of the DTD statement

```
  <!ATTLIST size unit CDATA #IMPLIED "cm">
```

Enclose element and type definitions of your schema inside an xsd:schema element:

```
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
  . . .
</xsd:schema>
```

Parsing an XML file with a schema is similar to parsing a file with a DTD, but with two differences:

1. You need to turn on support for namespaces, even if you don't use them in your XML files.

   ```
   factory.setNamespaceAware(true);
   ```

2. You need to prepare the factory for handling schemas, with the following magic incantation:

   ```
   final String JAXP_SCHEMA_LANGUAGE =
       "http://java.sun.com/xml/jaxp/properties/schemaLanguage"
   final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XM
   factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SC
   ```

3.4.3. A Practical Example

In this section, we work through a practical example that shows the use of XML in a realistic setting.

Suppose an application needs configuration data that specifies arbitrary objects, not just text strings. We provide two mechanisms for instantiating the object: with a constructor, and with a factory method. Here is how to make a Color object using a constructor:

```
<construct class="java.awt.Color">
  <int>55</int>
  <int>200</int>
  <int>100</int>
</construct>
```

Here is an example with a factory method:

```
<factory class="java.nio.file.Path" method="of">
  <string>logging.properties</string>
</factory>
```

If the factory method name is omitted, it defaults to getInstance.

As you can see, there are elements for describing strings and integers. We also support the boolean type, and other primitive types can be added in the same way.

Just to show off the syntax, there is a second mechanism for primitive types:

```
<value type="int">30</value>
```

A configuration is a sequence of entries. Each entry has an ID and an object:

```
<config>
  <entry id="background">
    <construct class="java.awt.Color">
      <value type="int">55</value>
      <value type="int">200</value>
      <value type="int">100</value>
    </construct>
  </entry>
  . . .
</config>
```

The parser checks that IDs are unique.

The program in Listing 3.2 shows how to parse a configuration file. A sample configuration is defined in Listing 3.3.

The program uses the schema instead of the DTD if you choose a file that contains the string -schema.

The DTD, shown in Listing 3.4, is straightforward.

Listing 3.5 contains the equivalent schema. In the schema, we can provide additional checking: an int or boolean element can only contain integer or boolean content. Note the use of the xsd:group construct to define parts of complex types that are used repeatedly.

This example is a typical use of XML. The XML format is robust enough to express complex relationships. The XML parser adds value by taking over the routine job of validity checking and supplying defaults.

Listing 3.2 **v2ch03/read/XMLReadDemo.java**

```
package v2ch03.read;

import module java.base;
import module java.xml;
import  org.w3c.dom.CharacterData;

/**
 * This program shows how to use an XML file to describe Java objects.
 */
class XMLReadDemo {
   void main(String[] args) throws Exception {
      String filename = args.length == 0 ? IO.readln("Input file: ") : args[0];

      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance
      factory.setValidating(true);

      if (filename.contains("-schema")) {
         factory.setNamespaceAware(true);
         final String JAXP_SCHEMA_LANGUAGE
            = "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
         final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema
         factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
      }

      factory.setIgnoringElementContentWhitespace(true);

      DocumentBuilder builder = factory.newDocumentBuilder();

      builder.setErrorHandler(new ErrorHandler() {
         public void warning(SAXParseException e) throws SAXException {
            System.err.println("Warning: " + e.getMessage());
         }

         public void error(SAXParseException e) throws SAXException {
            System.err.println("Error: " + e.getMessage());
```

```
            System.exit(0);
        }

        public void fatalError(SAXParseException e) throws SAXException {
            System.err.println("Fatal error: " + e.getMessage());
            System.exit(0);
        }
    });

    Document doc = builder.parse(filename);
    Map<String, Object> config = parseConfig(doc.getDocumentElement());
    IO.println(config);
}

Map<String, Object> parseConfig(Element e) throws ReflectiveOperationExc
    var result = new HashMap<String, Object>();
    NodeList children = e.getChildNodes();
    for (int i = 0; i < children.getLength(); i++) {
        var child = (Element) children.item(i);
        String name = child.getAttribute("id");
        Object value = parseObject((Element) child.getFirstChild());
        result.put(name, value);
    }
    return result;
}

Object parseObject(Element e) throws ReflectiveOperationException {
    String tagName = e.getTagName();
    if (tagName.equals("factory"))
        return parseFactory(e);
    else if (tagName.equals("construct"))
        return parseConstruct(e);
    else {
        String childData = ((CharacterData) e.getFirstChild()).getData();
        if (tagName.equals("int"))
            return Integer.valueOf(childData);
        else if (tagName.equals("boolean"))
            return Boolean.valueOf(childData);
        else
            return childData;
    }
}

Object parseFactory(Element e) throws ReflectiveOperationException {
```

```
      String className = e.getAttribute("class");
      String methodName = e.getAttribute("method");
      Object[] args = parseArgs(e.getChildNodes());
      Class<?>[] parameterTypes = getParameterTypes(args);
      Method method = Class.forName(className).getMethod(methodName, pa
      return method.invoke(null, args);
   }

   Object parseConstruct(Element e) throws ReflectiveOperationException {
      String className = e.getAttribute("class");
      Object[] args = parseArgs(e.getChildNodes());
      Class<?>[] parameterTypes = getParameterTypes(args);
      Constructor<?> constructor = Class.forName(className).getConstructor(p
      return constructor.newInstance(args);
   }

   Object[] parseArgs(NodeList elements) throws ReflectiveOperationExceptio
      var result = new Object[elements.getLength()];
      for (int i = 0; i < result.length; i++)
         result[i] = parseObject((Element) elements.item(i));
      return result;
   }

   Map<Class<?>, Class<?>> toPrimitive = Map.of(Integer.class, int.class, Boolea
         boolean.class);

   Class<?>[] getParameterTypes(Object[] args) {
      var result = new Class<?>[args.length];
      for (int i = 0; i < result.length; i++) {
         Class<?> cl = args[i].getClass();
         result[i] = toPrimitive.get(cl);
         if (result[i] == null) result[i] = cl;
      }
      return result;
   }
}
```

Listing 3.3 **v2ch03/read/config.xml**

```
<?xml version="1.0"?>
<!DOCTYPE config SYSTEM "config.dtd">
<config>
  <entry id="background">
```

```
    <construct class="java.awt.Color">
      <int>55</int>
      <int>200</int>
      <int>100</int>
    </construct>
  </entry>
  <entry id="currency">
    <factory class="java.util.Currency">
      <string>USD</string>
    </factory>
  </entry>
</config>
```

Listing 3.4 **v2ch03/read/config.dtd**

```
<!ELEMENT config (entry)*>

<!ELEMENT entry (string|int|boolean|construct|factory)>
<!ATTLIST entry id ID #IMPLIED>

<!ELEMENT construct (string|int|boolean|construct|factory)*>
<!ATTLIST construct class CDATA #IMPLIED>

<!ELEMENT factory (string|int|boolean|construct|factory)*>
<!ATTLIST factory class CDATA #IMPLIED>
<!ATTLIST factory method CDATA "getInstance">

<!ELEMENT string (#PCDATA)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Listing 3.5 **v2ch03/read/config.xsd**

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="config">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="entry" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:group ref="Object"/>
            <xsd:attribute name="id" type="xsd:ID"/>
          </xsd:complexType>
```

```
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="construct">
    <xsd:complexType>
      <xsd:group ref="Arguments"/>
      <xsd:attribute name="class" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="factory">
    <xsd:complexType>
      <xsd:group ref="Arguments"/>
      <xsd:attribute name="class" type="xsd:string"/>
      <xsd:attribute name="method" type="xsd:string" default="getInstance"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:group name="Object">
    <xsd:choice>
      <xsd:element ref="construct"/>
      <xsd:element ref="factory"/>
      <xsd:element name="string" type="xsd:string"/>
      <xsd:element name="int" type="xsd:int"/>
      <xsd:element name="boolean" type="xsd:boolean"/>
    </xsd:choice>
  </xsd:group>

  <xsd:group name="Arguments">
    <xsd:sequence>
      <xsd:group ref="Object" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:group>
</xsd:schema>
```

3.5. Locating Information with XPath

If you want to locate a specific piece of information in an XML document, it can be a bit of a hassle to navigate the nodes of the DOM tree. The XPath language makes it simple to access tree nodes. For example, suppose you have this XHTML document:

```
<html>
  <head>
    . . .
    <title>. . .</title>
    . . .
  </head>
  . . .
</html>
```

You can get the title text by evaluating the XPath expression

```
/html/head/title/text()
```

That's a lot simpler than the plain DOM approach:

1. Get the document root.
2. Get the first child and cast it as an Element.
3. Locate the title element among its children.
4. Get its first child and cast it as a CharacterData node.
5. Get its data.

An XPath can describe a set of nodes in an XML document. For example, the XPath

```
/html/body/form
```

describes the set of all form elements that are children of the body element in an XHTML file. You can select a particular element with the [] operator:

```
/html/body/form[1]
```

is the first form. (The index values start at 1.)

Use the @ operator to get attribute values. The XPath expression

```
/html/body/form[1]/@action
```

describes the action attribute of the first form. The XPath expression

```
/html/body/form/@action
```

describes all action attribute nodes of all form elements that are children of the body element.

There are a number of useful XPath functions. For example,

```
count(/html/body/form)
```

returns the number of form children of the body element. There are many more elaborate XPath expressions; see the specification at https://www.w3.org/TR/xpath or the online tutorial at https://www.zvon.org/xxl/XPathTutorial/General/examples.html.

To evaluate XPath expressions, first create an XPath object from an XPathFactory:

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

Then, call the evaluate method to evaluate XPath expressions:

```
String username = path.evaluate("/html/head/title/text()", doc);
```

You can use the same XPath object to evaluate multiple expressions.

This form of the evaluate method returns a string result. It is suitable for retrieving text, such as the text child of the title element in the preceding example. If an XPath expression yields multiple nodes, make a call such as the following:

```
XPathNodes result = path.evaluateExpression("/html/body/form", do
```

The XPathNodes class is similar to a NodeList, but it extends the Iterable interface, allowing you to use an enhanced for loop.

If the result is a single node, use one of the following calls:

```
Node node = path.evaluateExpression("/html/body/form[1]", doc, No
node = (Node) path.evaluate("/html/body/form[1]", doc, XPathConsta
```

If the result is a number, use:

```
int count = path.evaluateExpression("count(/html/body/form)", doc, 
count = ((Number) path.evaluate("count(/html/body/form)",
      doc, XPathConstants.NUMBER)).intValue();
```

You don't have to start the search at the document root; you can start at any node or node list. For example, if you have a node from a previous evaluation, you can call

```
String result = path.evaluate(expression, node);
```

If you do not know the result of evaluating an XPath expression (perhaps because it comes from a user), then call

```
XPathEvaluationResult<?> result = path.evaluateExpression(express
```

The expression result.type() is one of the constants

```
STRING
NODESET
NODE
NUMBER
BOOLEAN
```

of the XPathEvaluationResult.XPathResultType enumeration. Call result.value() to get the value.

The program in Listing 3.6 demonstrates evaluation of arbitrary XPath expressions. Specify the URL of an XML page and type expressions such as /html/body/h1[1]/text(). The results of the expressions are displayed.

Listing 3.6 **v2ch03/xpath/XPathDemo.java**

```
package v2ch03.xpath;

import module java.base;
import module java.xml;
```

```
/**
 * This program evaluates XPath expressions.
 */
class XPathDemo {
   void main(String[] args) throws Exception {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance
      DocumentBuilder builder = factory.newDocumentBuilder();

      // Avoid a delay in parsing an XHTML file--see the first note in Section 3.4.1
      builder.setEntityResolver(CatalogManager.catalogResolver(CatalogFeature
            Path.of("v2ch03", "xpath", "catalog.xml").toAbsolutePath().toUri()));

      XPathFactory xpfactory = XPathFactory.newInstance();
      XPath path = xpfactory.newXPath();

      String urlString;
      if (args.length == 0) {
         urlString = "https://www.horstmann.com/index.html";
         IO.println("Using " + urlString);
      }
      else
         urlString = args[0];
      Document doc = builder.parse(urlString);
      boolean done = false;
      while (!done) {
         IO.print("XPath expression (empty line to exit): ");
         String expression = IO.readln();
         if (expression.strip().isEmpty()) {
            done = true;
         }
         else {
            try {
               XPathEvaluationResult<?> result = path.evaluateExpression(expres
               if (result.type() == XPathEvaluationResult.XPathResultType.NODESE
                  for (Node n : (XPathNodes) result.value())
                     IO.println(description(n));
               }
               else if (result.type() == XPathEvaluationResult.XPathResultType.NO
                  IO.println((Node) result.value());
               else
                  IO.println(result.value());
            }
            catch (XPathExpressionException e) {
               IO.println(e.getMessage());
```

```
            }
         }
      }
   }

   String description(Node n) {
      return switch (n) {
         case Element _ -> "Element " + n.getNodeName();
         case Attr _ -> "Attribute " + n;
         default -> n.toString();
      };
   }
}
```

javax.xml.xpath.XPathFactory5.0

- static XPathFactory newInstance()
  returns an XPathFactory instance for creating XPath objects.
- XPath newXpath()
  constructs an XPath object for evaluating XPath expressions.

javax.xml.xpath.XPath5.0

- String evaluate(String expression, Object startingPoint)
  evaluates an expression, beginning at the given starting point. The starting point can be a node or node list. If the result is a node or node set, the returned string consists of the data of all text node children.
- Object evaluate(String expression, Object startingPoint, QName resultType)
  evaluates an expression, beginning at the given starting point. The starting point can be a node or node list. The resultType is one of the constants STRING, NODE, NODESET, NUMBER, or BOOLEAN in the XPathConstants class. The return value is a String, Node, NodeList, Number, or Boolean.
- <T> T evaluateExpression(String expression, Object item, Class<T> type)9

evaluates the given expression and yields the result as a value of the given type.

- XPathEvaluationResult<?> evaluateExpression(String expression, InputSource source)9
  evaluates the given expression.

javax.xml.xpath.XPathEvaluationResult<T>9

- XPathEvaluationResult.XPathResultType type()
  returns one of the enumeration constants STRING, NODESET, NODE, NUMBER, BOOLEAN.
- T value()
  returns the result value.

3.6. Using Namespaces

The Java language uses packages to avoid name clashes. Programmers can use the same name for different classes as long as they aren't in the same package. XML has a similar namespace mechanism for element and attribute names.

A namespace is identified by a Uniform Resource Identifier (URI). Here are three examples:

```
http://www.w3.org/2001/XMLSchema
uuid:1c759aed-b748-475c-ab68-10679700c4f2
urn:com:books-r-us
```

The HTTP URL form is the most common. Note that the URL is just used as an identifier string, not as a locator for a document. For example, the namespace identifiers

```
http://www.horstmann.com/corejava
http://www.horstmann.com/corejava/index.html
```

denote different namespaces, even though a web server would serve the same document for both URLs.

There need not be any document at a namespace URL—the XML parser doesn't attempt to find anything at that location. However, as a help to programmers who encounter a possibly unfamiliar namespace, it is customary to place a document explaining the purpose of the namespace at the URL location. For example, if you point your browser to the namespace URL for the XML Schema namespace (http://www.w3.org/2001/XMLSchema), you will find a document describing the XML Schema standard.

Why use HTTP URLs for namespace identifiers? It is easy to ensure that they are unique. If you choose a real URL, the host part's uniqueness is guaranteed by the domain name system. Your organization can then arrange for the uniqueness of the remainder of the URL. This is the same rationale that underlies the use of reversed domain names in Java package names.

Of course, although long namespace identifiers are good for uniqueness, you don't want to deal with long identifiers any more than you have to. In the Java programming language, you use the import mechanism to specify the long names of packages, and then use just the short class names. In XML, there is a similar mechanism:

```
<tagName xmlns="namespaceURI">
  children
</tagName>
```

The element and its children are now part of the given namespace.

A child can provide its own namespace, for example:

```
<tagName xmlns="namespaceURI">
  <childTagName xmlns="childNamespaceURI">
    grandchildren
  </childTagName>
  more children
</tagName>
```

Then the first child and the grandchildren are part of the

second namespace.

This simple mechanism works well if you need only a single namespace or if the namespaces are naturally nested. Otherwise, you will want to use a second mechanism that has no analog in Java. You can have a prefix for a namespace—a short identifier that you choose for a particular document. Here is a typical example—the xsd prefix in an XML Schema file:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="config"/>
  . . .
</xsd:schema>
```

The attribute

```
xmlns:prefix="namespaceURI"
```

defines a namespace and a prefix. In our example, the prefix is the string xsd. Thus, xsd:schema really means schema in the namespace http://www.w3.org/2001/XMLSchema.

Only child elements inherit the namespace of their parent. Attributes without an explicit prefix are never part of a namespace. Consider this contrived example:

```
<configuration xmlns="http://www.horstmann.com/corejava"
  xmlns:si="http://www.bipm.fr/enus/3_SI/si.html">
  <size value="210" si:unit="mm"/>
  . . .
</configuration>
```

In this example, the elements configuration and size are part of the namespace with URI http://www.horstmann.com/corejava. The attribute si:unit is part of the namespace with URI http://www.bipm.fr/enus/3_SI/si.html. However, the attribute value is not part of any namespace.

You can control how the parser deals with namespaces. By default, the DOM parser of the Java XML API is not namespace-aware.

To turn on namespace handling, call the setNamespaceAware method of the DocumentBuilderFactory:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newIns
factory.setNamespaceAware(true);
```

Now, all builders the factory produces support namespaces.

Alternatively, since Java 13, call

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newNS
```

Each node created by a builder has three properties:

- The qualified name, with a prefix, returned by getNodeName, getTagName, and so on
- The namespace URI, returned by the getNamespaceURI method
- The local name, without a prefix or a namespace, returned by the getLocalName method

Here is an example. Suppose the parser sees the following element:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

It then reports the following:

- Qualified name = xsd:schema
- Namespace URI = http://www.w3.org/2001/XMLSchema
- Local name = schema

If namespace awareness is turned off, getNamespaceURI and getLocalName return null.

---

org.w3c.dom.Node1.4

- String getLocalName()
  returns the local name (without prefix), or null if the parser is not namespace-aware.
- String getNamespaceURI()
  returns the namespace URI, or null if the node is not part of a namespace or if the parser is not namespace-aware.

In the Java API, namespaces interact poorly with XPath expressions. You need to manually install a "namespace context" to resolve namespace prefixes. Moreover, Java still uses XPath 1.0, so you can't use the *:*tagName* syntax of XPath 2.0. The following context uses an _ instead:

```
path.setNamespaceContext(new NamespaceContext() {
   public String getNamespaceURI(String prefix) {
      return doc.lookupNamespaceURI(prefix.equals("_") ? null
   }
   public Iterator<String> getPrefixes(String val) { return null;
   public String getPrefix(String uri) { return null; }
});
```

Use _ for the default namespace and prefixes for the others, such as //_:div/svg:svg.

---

javax.xml.parsers.DocumentBuilderFactory1.4

- static DocumentBuilderFactory newNSInstance()13
  yields a factory whose builders are namespace-aware.
- boolean isNamespaceAware()
- void setNamespaceAware(boolean value)
  get or set the namespaceAware property of the factory. If set to true, the parsers that this factory generates are namespace-aware.

3.7. Streaming Parsers

The DOM parser reads an XML document in its entirety into a tree data structure. For most practical applications, DOM works fine. However, it can be inefficient if the document is large and if your processing algorithm is simple enough that you can analyze nodes on the fly, without having to see all of the tree structure. In these cases, you should use a streaming parser.

In the following sections, we discuss two streaming parsers supplied by the Java API. The SAX parser uses event callbacks, and the StAX parser provides an iterator through the parsing events. The latter is usually a bit more convenient.

3.7.1. Using the SAX Parser

The SAX parser reports events as it parses the components of the XML input, but it does not store the document in any way—it is up to the event handlers to build a data structure. In fact, the DOM parser is built on top of the SAX parser. It builds the DOM tree as it receives the parser events.

Whenever you use a SAX parser, you need a handler that defines the event actions for the various parse events. The ContentHandler interface defines several callback methods that the parser executes as it parses the document. Here are the most important ones:

- startElement and endElement are called each time a start tag or end tag is encountered.
- characters is called whenever character data are encountered.
- startDocument and endDocument are called once each, at the start and the end of the document.

For example, when parsing the fragment

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

the parser makes the following callbacks:

1. startElement, element name: font
2. startElement, element name: name
3. characters, content: Helvetica
4. endElement, element name: name
5. startElement, element name: size, attributes: units="pt"
6. characters, content: 36
7. endElement, element name: size
8. endElement, element name: font

Your handler needs to override these methods and have them carry out whatever action you want to carry out as you parse the file. The program at the end of this section prints all links of the form <a href=". . ."> in an HTML file. It simply overrides the startElement method of the handler to check for elements with name a and an attribute with name href. This is potentially useful for implementing a "web crawler"—a program that reaches more and more web pages by following links.

HTML doesn't have to be valid XML, and most web pages deviate so much from proper XML that the example programs will not be able to parse them. However, the pages on my web site https://horstmann.com are written in XHTML (an HTML dialect that is proper XML). You can use those pages to test the example program. For example, if you run

```
java v2ch03.sax.SAXDemo https://horstmann.com/corejava/inc
```

you will see a list of the URLs of all links on that page.

The sample program is a good example for the use of SAX. We don't care at all in which context the a elements occur, and there is no need to store a tree structure.

Here is how you get a SAX parser:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

You can now process a document:

```
parser.parse(source, handler);
```

Here, source can be a file, URL string, or input stream. The handler belongs to a subclass of DefaultHandler. The DefaultHandler class defines do-nothing methods for the four interfaces:

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

The example program defines a handler that overrides the startElement method of the ContentHandler interface to watch out for a elements with an href attribute:

```
var handler = new DefaultHandler() {
      public void startElement(String namespaceURI, String lname, St
            Attributes attrs) throws SAXException {
         if (lname.equalsIgnoreCase("a") && attrs != null) {
            String href = attrs.getValue("href");
            if (href != null) IO.println(href);
         }
      }
   };
```

The startElement method has three parameters that describe the element name. The qname parameter reports the qualified name of the form prefix:localname. If namespace processing is turned on, then the namespaceURI and lname parameters provide the namespace and local name (without the namespace).

As with the DOM parser, namespace processing is turned off by default. To activate namespace processing, call the setNamespaceAware method of the factory class. Alternatively, as of Java 13, make a namespace-aware factory with newNSInstance:

```
SAXParserFactory factory = SAXParserFactory.newNSInstance();
SAXParser saxParser = factory.newSAXParser();
```

In this program, we cope with another common issue. An XHTML file starts with a tag that contains a DTD reference, and the parser will want to load it. Understandably, the W3C isn't too happy to serve billions of copies of files such as www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd. At one point, they refused altogether, but at the time of this writing, they serve the DTD at a glacial pace. If you don't need to validate the document, just call

```
factory.setFeature(
   "http://apache.org/xml/features/nonvalidating/load-external-dtd", f
```

Listing 3.7 contains the code for the web crawler program. Later in this chapter, you will see another interesting use of SAX. An easy way of turning a non-XML data source into XML is to report the SAX events that an XML parser would report. See Section 3.9 for details.

Listing 3.7 **v2ch03/sax/SAXDemo.java**

```
package v2ch03.sax;

import module java.base;
import module java.xml;
import org.xml.sax.Attributes;

/**
 * This program demonstrates how to use a SAX parser. The program prints all
 * XHTML web page.
 */
class SAXDemo {
   void main(String[] args) throws Exception {
      String url;
      if (args.length == 0) {
         url = "https://horstmann.com/corejava/index.html";
         IO.println("Using " + url);
      }
      else
         url = args[0];

      var handler = new DefaultHandler() {
```

```
        public void startElement(String namespaceURI, String lname, String qna
            Attributes attrs) {
          if (lname.equals("a") && attrs != null) {
            String href = attrs.getValue("href");
            if (href != null) IO.println(href);
          }
        }
      };

      SAXParserFactory factory = SAXParserFactory.newInstance();
      factory.setNamespaceAware(true);
      factory.setFeature("http://apache.org/xml/features/nonvalidating/load-exte
          false);
      SAXParser saxParser = factory.newSAXParser();
      InputStream in = new URI(url).toURL().openStream();
      saxParser.parse(in, handler);
    }
}
```

javax.xml.parsers.SAXParserFactory1.4

- static SAXParserFactory newInstance()
- static SAXParserFactory newNSInstance()13
  return an instance of the SAXParserFactory class. The second method yields a namespace-aware instance.
- SAXParser newSAXParser()
  returns an instance of the SAXParser class.
- boolean isNamespaceAware()
- void setNamespaceAware(boolean value)
  get or set the namespaceAware property of the factory. If set to true, the parsers that this factory generates are namespace-aware.
- boolean isValidating()
- void setValidating(boolean value)
  get or set the validating property of the factory. If set to true, the parsers that this factory generates validate their input.

javax.xml.parsers.SAXParser1.4

- void parse(File f, DefaultHandler handler)
- void parse(String url, DefaultHandler handler)
- void parse(InputStream in, DefaultHandler handler)
  parse an XML document from the given file, URL, or input stream and report parse events to the given handler.

org.xml.sax.ContentHandler1.4

- void startDocument()
- void endDocument()
  are called at the start or the end of the document.
- void startElement(String uri, String lname, String qname, Attributes attr)
- void endElement(String uri, String lname, String qname)
  are called at the start or the end of an element. If the parser is namespace-aware, they report the URI of the namespace, the local name without prefix, and the qualified name with prefix.
- void characters(char[] data, int start, int length)
  is called when the parser reports character data.

org.xml.sax.Attributes1.4

- int getLength()
  returns the number of attributes stored in this attribute collection.
- String getLocalName(int index)
  returns the local name (without prefix) of the attribute with the given index, or the empty string if the parser is not namespace-aware.
- String getURI(int index)
  returns the namespace URI of the attribute with the given index, or the empty string if the node is not part of a namespace or if the parser is not namespace-aware.
- String getQName(int index)
  returns the qualified name (with prefix) of the attribute with the given index, or the empty string if the qualified

name is not reported by the parser.
- String getValue(int index)
- String getValue(String qname)
- String getValue(String uri, String lname)
  return the attribute value from a given index, qualified name, or namespace URI + local name. Return null if the value doesn't exist.

3.7.2. Using the StAX Parser

The StAX parser is a "pull parser." Instead of installing an event handler, you simply iterate through the events, using this basic loop:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext()) {
   int event = parser.next();
   Call parser methods to obtain event details
}
```

For example, when parsing the fragment

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

the parser yields the following events:

1. START_ELEMENT, element name: font
2. CHARACTERS, content: white space
3. START_ELEMENT, element name: name
4. CHARACTERS, content: Helvetica
5. END_ELEMENT, element name: name
6. CHARACTERS, content: white space
7. START_ELEMENT, element name: size
8. CHARACTERS, content: 36

9. END_ELEMENT, element name: size
10. CHARACTERS, content: white space
11. END_ELEMENT, element name: font

To analyze the attribute values, call the appropriate methods of the XMLStreamReader class. For example,

```
String units = parser.getAttributeValue(null, "units");
```

gets the units attribute of the current element.

By default, namespace processing is enabled. You can deactivate it by modifying the factory:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false)
```

Listing 3.8 contains the code for the web crawler program implemented with the StAX parser. As you can see, the code is simpler than the equivalent SAX code because you don't have to worry about event handling.

Listing 3.8 **v2ch03/stax/StAXDemo.java**

```
package v2ch03.stax;

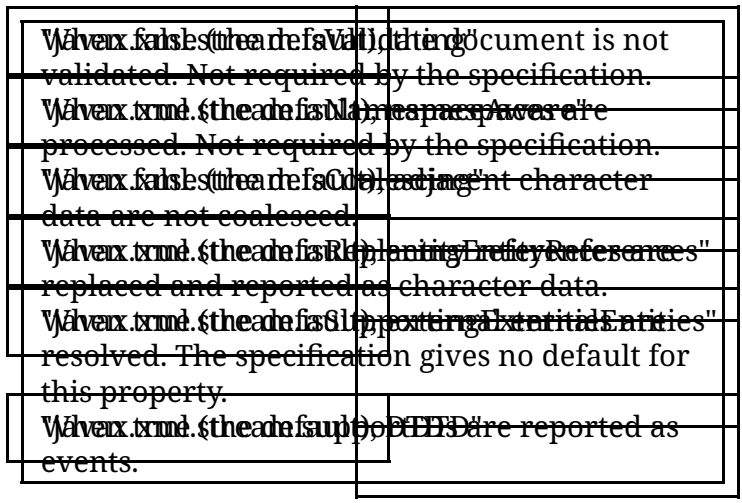
import module java.base;
import module java.xml;

/**
 * This program demonstrates how to use a StAX parser. The program prints al
 * XHTML web page.
 */
class StAXDemo {
   void main(String[] args) throws Exception {
      String urlString;
      if (args.length == 0) {
         urlString = "https://www.horstmann.com/index.html";
         IO.println("Using " + urlString);
      }
      else
```

```
      urlString = args[0];
    InputStream in = new URI(urlString).toURL().openStream();
    XMLInputFactory factory = XMLInputFactory.newInstance();
    XMLStreamReader parser = factory.createXMLStreamReader(in);
    while (parser.hasNext()) {
      int event = parser.next();
      if (event == XMLStreamConstants.START_ELEMENT) {
        if (parser.getLocalName().equals("a")) {
          String href = parser.getAttributeValue(null, "href");
          if (href != null) IO.println(href);
        }
      }
    }
  }
}
```

javax.xml.stream.XMLInputFactory 6

- static XMLInputFactory newInstance()
  returns an instance of the XMLInputFactory class.
- void setProperty(String name, Object value)
  sets a property for this factory, or throws an IllegalArgumentException if the property is not supported or cannot be set to the given value. The JDK implementation supports the following Boolean-valued properties:



- XMLStreamReader createXMLStreamReader(InputStream

in)
- XMLStreamReader createXMLStreamReader(InputStream in, String characterEncoding)
- XMLStreamReader createXMLStreamReader(Reader in)
- XMLStreamReader createXMLStreamReader(Source in)
create a parser that reads from the given stream, reader, or JAXP source.

javax.xml.stream.XMLStreamReader6

- boolean hasNext()
returns true if there is another parse event.
- int next()
sets the parser state to the next parse event and returns one of the following constants: START_ELEMENT, END_ELEMENT, CHARACTERS, START_DOCUMENT, END_DOCUMENT, CDATA, COMMENT, SPACE (ignorable whitespace), PROCESSING_INSTRUCTION, ENTITY_REFERENCE, DTD.
- boolean isStartElement()
- boolean isEndElement()
- boolean isCharacters()
- boolean isWhiteSpace()
return true if the current event is a start element, end element, character data, or whitespace.
- QName getName()
- String getLocalName()
get the name of the element in a START_ELEMENT or END_ELEMENT event.
- String getText()
returns the characters of a CHARACTERS, COMMENT, or CDATA event, the replacement value for an ENTITY_REFERENCE, or the internal subset of a DTD.
- int getAttributeCount()
- QName getAttributeName(int index)
- String getAttributeLocalName(int index)
- String getAttributeValue(int index)
get the attribute count and the names and values of the attributes, provided the current event is START_ELEMENT.

- String getAttributeValue(String namespaceURI, String name)
  gets the value of the attribute with the given name, provided the current event is START_ELEMENT. If namespaceURI is null, the namespace is not checked.

3.8. Generating XML Documents

You now know how to write Java programs that read XML. Let us now turn to the opposite process: producing XML output. Of course, you could write an XML file simply by making a sequence of print calls, printing the elements, attributes, and text content, but that would not be a good idea. The code is rather tedious, and you can easily make mistakes if you don't pay attention to special symbols (such as " or <) in the attribute values and text content.

A better approach is to build up a DOM tree with the contents of the document and then write out the tree contents. The following sections discuss the details.

3.8.1. Documents without Namespaces

To build a DOM tree, you start out with an empty document. You can get an empty document by calling the newDocument method of the DocumentBuilder class:

```
Document doc = builder.newDocument();
```

Use the createElement method of the Document class to construct the elements of your document:

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Use the createTextNode method to construct text nodes:

```
Text textNode = doc.createTextNode(textContents);
```

Add the root element to the document, and add the child nodes to their parents:

```
doc.appendChild(rootElement);
rootElement.appendChild(childElement);
childElement.appendChild(textNode);
```

As you build up the DOM tree, you may also need to set element attributes. Simply call the setAttribute method of the Element class:

```
rootElement.setAttribute(name, value);
```

3.8.2. Documents with Namespaces

If you use namespaces, the procedure for creating a document is slightly different.

First, set the builder factory to be namespace-aware, then create the builder:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newIns
factory.setNamespaceAware(true);
builder = factory.newDocumentBuilder();
```

Then use createElementNS instead of createElement to create any nodes:

```
String namespace = "http://www.w3.org/2000/svg";
Element rootElement = doc.createElementNS(namespace, "svg");
```

If your node has a qualified name with a namespace prefix, then any necessary xmlns-prefixed attributes are created automatically. For example, if you need SVG inside XHTML, you can construct an element like this:

```
Element svgElement = doc.createElement(namespace, "svg:svg")
```

When the element is written, it turns into

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
```

If you need to set element attributes whose names are in a namespace, use the setAttributeNS method of the Element class:

```
rootElement.setAttributeNS(namespace, qualifiedName, value);
```

3.8.3. Writing Documents

Somewhat curiously, it is not so easy to write a DOM tree to an output stream. The easiest approach is to use the Extensible Stylesheet Language Transformations (XSLT) API. For more information about XSLT, turn to Section 3.9. Right now, consider the code that follows a magic incantation to produce XML output.

We apply the do-nothing transformation to the document and capture its output. To include a DOCTYPE node in the output, we also need to set the SYSTEM and PUBLIC identifiers as output properties.

```
// construct the do-nothing transformation
Transformer t = TransformerFactory.newInstance().newTransforme
// set output properties to get a DOCTYPE node
t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, systemIdentifie
t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, publicIdentifier)
// set indentation
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2"
// apply the do-nothing transformation and send the output to a file
t.transform(new DOMSource(doc), new StreamResult(new FileOutpu
```

Another approach is to use the LSSerializer interface. To get an instance, you have to use the following magic incantation:

```
DOMImplementation impl = doc.getImplementation();
var implLS = (DOMImplementationLS) impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();
```

If you want spaces and line breaks, set this flag:

```
ser.getDomConfig().setParameter("format-pretty-print", true);
```

Then it's simple enough to convert a document to a string:

```
String str = ser.writeToString(doc);
```

If you want to write the output directly to a file, you need an LSOutput:

```
LSOutput out = implLS.createLSOutput();
out.setEncoding("UTF-8");
out.setByteStream(Files.newOutputStream(path));
ser.write(doc, out);
```

javax.xml.parsers.DocumentBuilder 1.4

- Document newDocument()
  returns an empty document.

org.w3c.dom.Document 1.4

- Element createElement(String name)
- Element createElementNS(String uri, String qname)
  create an element with the given name.
- Text createTextNode(String data)
  creates a text node with the given data.

org.w3c.dom.Node 1.4

- Node appendChild(Node child)
  appends a node to the list of children of this node. Returns the appended node.

org.w3c.dom.Element 1.4

- void setAttribute(String name, String value)

- void setAttributeNS(String uri, String qname, String value)
  set the attribute with the given name to the given value. If the qualified name has an alias prefix, then uri must not be null.

javax.xml.transform.TransformerFactory1.4

- static TransformerFactory newInstance()
  returns an instance of the TransformerFactory class.
- Transformer newTransformer()
  returns an instance of the Transformer class that carries out an identity (do-nothing) transformation.

javax.xml.transform.Transformer1.4

- void setOutputProperty(String name, String value)
  sets an output property. See https://www.w3.org/TR/xslt#output for a listing of the standard output properties. The most useful ones are shown here:

The public ID to be used in the DOCTYPE declaration	
The system ID to be used in the DOCTYPE declaration	
"yes" or "no"	
"xml", "html", "text", or a custom string	

- void transform(Source from, Result to)
  transforms an XML document.

javax.xml.transform.dom.DOMSource1.4

- DOMSource(Node n)
  constructs a source from the given node. Usually, n is a document node.

javax.xml.transform.stream.StreamResult1.4

- StreamResult(File f)
- StreamResult(OutputStream out)
- StreamResult(Writer out)
- StreamResult(String systemID)
  construct a stream result from a file, stream, writer, or system ID (usually a relative or absolute URL).

3.8.4. Writing an XML Document with StAX

In the preceding section, you saw how to produce an XML document by writing a DOM tree. If you have no other use for the DOM tree, that approach is not very efficient.

The StAX API lets you write an XML tree directly. Construct an XMLStreamWriter from an OutputStream:

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

To produce the XML header, call

```
writer.writeStartDocument();
```

Then call

```
writer.writeStartElement(name);
```

Add attributes by calling

```
writer.writeAttribute(name, value);
```

Now you can add child elements by calling writeStartElement again, or write characters with

```
writer.writeCharacters(text);
```

When you have written all child nodes, call

```
writer.writeEndElement();
```

This causes the current element to be closed.

To write an element without children (such as), use the call

```
writer.writeEmptyElement(name);
```

Finally, at the end of the document, call

```
writer.writeEndDocument();
```

This call closes any open elements.

You still need to close the XMLStreamWriter, and you need to do it manually since the XMLStreamWriter interface does not extend the AutoCloseable interface.

As with the DOM/XSLT approach, you don't have to worry about escaping characters in attribute values and character data. However, it is possible to produce malformed XML, such as a document with multiple root nodes. Also, the current version of StAX has no support for producing indented output.

Listing 3.9 demonstrates two ways of producing XML: by constructing and saving a DOM tree, and by directly writing the XML with the StAX API.

The program draws a modernist painting—a random set of colored rectangles (see Figure 3.3). To save our masterpiece, we use the Scalable Vector Graphics (SVG) format. SVG is an XML format to describe complex graphics in a device-independent fashion. You can find more information about SVG at https://www.w3.org/Graphics/SVG. To view SVG files, simply use any modern browser.



Figure 3.3: Generating modern art

We don't need to go into details about SVG; for our purposes, we just need to know how to express a set of colored rectangles. Here is a sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
  "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.d
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="150"
  <rect x="231" y="61" width="9" height="12" fill="#6e4a13"/>
  <rect x="107" y="106" width="56" height="5" fill="#c406be"/>
  . . .
</svg>
```

As you can see, each rectangle is described as a rect node. The position, width, height, and fill color are attributes. The fill color is an RGB value in hexadecimal.

SVG uses attributes heavily. In fact, some attributes are quite complex. For example, here is a path element:

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

The M denotes a "moveto" command, L is "lineto," and z is "closepath" (!). Apparently, the designers of this data format didn't have much confidence in using XML for structured data. In your own XML formats, you might want to use elements instead of complex attributes. Or, if you find that XML is too verbose for your purpose, choose a different format.

Listing 3.9 **v2ch03/write/XMLWriteDemo.java**

```
package v2ch03.write;

import module java.base;
import module java.xml;

/**
 * This program shows how to write an XML file. It produces modern art in SV
 */
class XMLWriteDemo {
   void main(String[] args) throws Exception {
      Document doc = newDrawing(600, 400);
      writeDocument(doc, "drawing1.svg");
      writeNewDrawing(600, 400, "drawing2.svg");
   }

   RandomGenerator generator = RandomGenerator.getDefault();

   /**
    * Creates a new random drawing.
    * @param drawingWidth the width of the drawing in pixels
    * @param drawingHeight the height of the drawing in pixels
    * @return the DOM tree of the SVG document
    */
   Document newDrawing(int drawingWidth, int drawingHeight)
         throws ParserConfigurationException {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance
      factory.setNamespaceAware(true);
      DocumentBuilder builder = factory.newDocumentBuilder();
      ;
      String namespace = "http://www.w3.org/2000/svg";
      Document doc = builder.newDocument();
```

```
    Element svgElement = doc.createElementNS(namespace, "svg");
    doc.appendChild(svgElement);
    svgElement.setAttribute("width", "" + drawingWidth);
    svgElement.setAttribute("height", "" + drawingHeight);
    int n = 10 + generator.nextInt(20);
    for (int i = 1; i <= n; i++) {
        int x = generator.nextInt(drawingWidth);
        int y = generator.nextInt(drawingHeight);
        int width = generator.nextInt(drawingWidth - x);
        int height = generator.nextInt(drawingHeight - y);
        int r = generator.nextInt(256);
        int g = generator.nextInt(256);
        int b = generator.nextInt(256);

        Element rectElement = doc.createElementNS(namespace, "rect");
        rectElement.setAttribute("x", "" + x);
        rectElement.setAttribute("y", "" + y);
        rectElement.setAttribute("width", "" + width);
        rectElement.setAttribute("height", "" + height);
        rectElement.setAttribute("fill", "#%02x%02x%02x".formatted(r, g, b));
        svgElement.appendChild(rectElement);
    }
    return doc;
}

/**
 * Saves a document using DOM/XSLT.
 * @param doc the document to be written
 * @param filename the name of the destination file
 */
void writeDocument(Document doc, String filename)
        throws TransformerException, IOException {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
            "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd"
    t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "-//W3C//DTD SVG 200
    t.setOutputProperty(OutputKeys.INDENT, "yes");
    t.setOutputProperty(OutputKeys.METHOD, "xml");
    t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
    t.transform(new DOMSource(doc),
            new StreamResult(Files.newOutputStream(Path.of(filename))));
}

/**
```

```
   * Uses StAX to write an SVG document with a random drawing.
   * @param drawingWidth the width of the drawing in pixels
   * @param drawingHeight the width of the drawing in pixels
   * @param filename the name of the destination file
   */
   void writeNewDrawing(int drawingWidth, int drawingHeight, String filenar
         throws XMLStreamException, IOException {
      XMLOutputFactory factory = XMLOutputFactory.newInstance();
      XMLStreamWriter writer = factory
            .createXMLStreamWriter(Files.newOutputStream(Path.of(filename)))
      writer.writeStartDocument();
      writer.writeDTD("""
            <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
              "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd
            """);
      writer.writeStartElement("svg");
      writer.writeDefaultNamespace("http://www.w3.org/2000/svg");
      writer.writeAttribute("width", "" + drawingWidth);
      writer.writeAttribute("height", "" + drawingHeight);
      int n = 10 + generator.nextInt(20);
      for (int i = 1; i <= n; i++) {
         int x = generator.nextInt(drawingWidth);
         int y = generator.nextInt(drawingHeight);
         int width = generator.nextInt(drawingWidth - x);
         int height = generator.nextInt(drawingHeight - y);
         int r = generator.nextInt(256);
         int g = generator.nextInt(256);
         int b = generator.nextInt(256);
         writer.writeEmptyElement("rect");
         writer.writeAttribute("x", "" + x);
         writer.writeAttribute("y", "" + y);
         writer.writeAttribute("width", "" + width);
         writer.writeAttribute("height", "" + height);
         writer.writeAttribute("fill", "#%02x%02x%02x".formatted(r, g, b));
      }
      writer.writeEndDocument(); // closes svg element
   }
}
```

javax.xml.stream.XMLOutputFactory6

- static XMLOutputFactory newInstance()
  returns an instance of the XMLOutputFactory class.
- XMLStreamWriter createXMLStreamWriter(OutputStream

stream)
- XMLStreamWriter createXMLStreamWriter(OutputStream stream, String encoding)
- XMLStreamWriter createXMLStreamWriter(Writer writer)
- XMLStreamWriter createXMLStreamWriter(Result result)
  create a writer that writes to the given stream, writer, or JAXP result.

javax.xml.stream.XMLStreamWriter6

- void writeStartDocument()
- void writeStartDocument(String xmlVersion)
- void writeStartDocument(String encoding, String xmlVersion)
  write the XML processing instruction at the top of the document. Note that the encoding parameter is only used to write the attribute. It does not set the character encoding of the output.
- void setDefaultNamespace(String namespaceURI)
- void setPrefix(String prefix, String namespaceURI)
  set the default namespace or the namespace associated with a prefix. The declaration is scoped to the current element or, if no element has been written, to the document root.
- void writeStartElement(String localName)
- void writeStartElement(String namespaceURI, String localName)
  write a start tag, replacing the namespaceURI with the associated prefix.
- void writeEndElement()
  closes the current element.
- void writeEndDocument()
  closes all open elements.
- void writeEmptyElement(String localName)
- void writeEmptyElement(String namespaceURI, String localName)
  write a self-closing tag, replacing the namespaceURI with the associated prefix.

- void writeAttribute(String localName, String value)
- void writeAttribute(String namespaceURI, String localName, String value)
  write an attribute for the current element, replacing the namespaceURI with the associated prefix.
- void writeCharacters(String text)
  writes character data.
- void writeCData(String text)
  writes a CDATA block.
- void writeDTD(String dtd)
  writes the dtd string, which is assumed to contain a DOCTYPE declaration.
- void writeComment(String comment)
  writes a comment.
- void close()
  closes this writer.

3.9. XSL Transformations

The XSL Transformations (XSLT) mechanism allows you to specify rules for transforming XML documents into other formats, such as plain text, XHTML, or any other XML format. XSLT is commonly used to translate from one machine-readable XML format to another, or to translate XML into a presentation format for human consumption.

You need to provide an XSLT stylesheet that describes the conversion of XML documents into some other format. An XSLT processor reads an XML document and the stylesheet and produces the desired output (see Figure 3.4).

.

Figure 3.4: Applying XSL transformations

The XSLT specification is quite complex, and entire books have been written on the subject. We can't possibly discuss all the features of XSLT, so we will just work through a representative

example. You can find more information in the book Essential XML by Don Box et al. The XSLT specification is available at https://www.w3.org/TR/xslt.

Suppose we want to transform XML files with employee records into HTML documents. Consider this input file:

```
<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000.0</salary>
    <hiredate year="1987" month="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000.0</salary>
    <hiredate year="1989" month="10" day="1"/>
  </employee>
  <employee>
    <name>Tony Tester</name>
    <salary>40000.0</salary>
    <hiredate year="1990" month="3" day="15"/>
  </employee>
</staff>
```

The desired output is an HTML table:

```
<table border="1">
<tr>
<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>
</tr>
<tr>
<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>
</tr>
</table>
```

A stylesheet with transformation templates has this form:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
  <xsl:output method="html"/>
  template1

  template2
  . . .
</xsl:stylesheet>
```

In our example, the xsl:output element specifies the method as HTML. Other valid method settings are xml and text.

Here is a typical template:

```
<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>
```

The value of the match attribute is an XPath expression. The template states: Whenever you see a node in the XPath set /staff/employee, do the following:

1. Emit the string <tr>.
2. Keep applying templates as you process its children.
3. Emit the string </tr> after you are done with all children.

In other words, this template generates the HTML table row markers around every employee record.

The XSLT processor starts processing by examining the root element. Whenever a node matches one of the templates, it applies the template. (If multiple templates match, the best matching one is used; see the specification at https://www.w3.org/TR/xslt for the gory details.) If no template matches, the processor carries out a default action. For text nodes, the default is to include the contents in the output. For elements, the default action is to create no output but to keep processing the children.

Here is a template for transforming name nodes in an employee file:

```
<xsl:template match="/staff/employee/name">
  <td><xsl:apply-templates/></td>
</xsl:template>
```

As you can see, the template produces the <td>. . .</td> delimiters, and it asks the processor to recursively visit the children of the name element. There is just one child—the text node. When the processor visits that node, it emits the text contents (provided, of course, that there is no other matching template).

You have to work a little harder if you want to copy attribute values into the output. Here is an example:

```
<xsl:template match="/staff/employee/hiredate">
  <td><xsl:value-of select="@year"/>-<xsl:value-of
  select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

When processing a hiredate node, this template emits

1. The string <td>
2. The value of the year attribute
3. A hyphen
4. The value of the month attribute
5. A hyphen
6. The value of the day attribute
7. The string </td>

The xsl:value-of statement computes the string value of a node set. The node set is specified by the XPath value of the select attribute. In this case, the path is relative to the currently processed node. The node set is converted to a string by concatenation of the string values of all nodes. The string value of an attribute node is its value. The string value of a text node is its contents. The string value of an element node is the concatenation of the string values of its child nodes (but not its attributes).

Listing 3.10 contains the stylesheet for turning an XML file with employee records into an HTML table.

Listing 3.11 shows a different set of transformations. The input is the same XML file, and the output is plain text in the familiar property file format:

```
employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15
```

That example uses the position() function which yields the position of the current node as seen from its parent. We thus get an entirely different output simply by switching the stylesheet. This means you can safely use XML to describe your data; if some applications need the data in another format, just use XSLT to generate the alternative format.

It is simple to generate XSL transformations on the Java platform. Set up a transformer factory for each stylesheet. Then, get a transformer object and tell it to transform a source to a result:

```
var styleSheet = new File(filename);
var styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance().newTransforme
t.transform(source, result);
```

The arguments of the transform method are objects of classes that implement the Source and Result interfaces. Several classes implement the Source interface:

```
DOMSource
SAXSource
StAXSource
```

StreamSource

You can construct a StreamSource from a file, stream, reader, or URL, and a DOMSource from the node of a DOM tree. For example, in the preceding section, we invoked the identity transformation as

```
t.transform(new DOMSource(doc), result);
```

In our example program, we do something slightly more interesting. Instead of starting out with an existing XML file, we produce a SAX XML reader that gives the illusion of parsing an XML file by emitting appropriate SAX events. Actually, our XML reader reads a flat file, as described in Chapter 1. The input file looks like this:

```
Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15
```

Our XML reader generates SAX events as it processes the input. Here is a part of the parse method of the EmployeeReader class that implements the XMLReader interface:

```
var attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
boolean done = false;
while (!done) {
   String line = in.readLine();
   if (line == null) done = true;
   else {
      handler.startElement("", "employee", "employee", attributes);
      var tokenizer = new StringTokenizer(line, "|");
      handler.startElement("", "name", "name", attributes);
      String s = tokenizer.nextToken();
      handler.characters(s.toCharArray(), 0, s.length());
      handler.endElement("", "name", "name");
      . . .
      handler.endElement("", "employee", "employee");
   }
```

```
}
handler.endElement("", rootElement, rootElement);
handler.endDocument();
```

The SAXSource for the transformer is constructed from the XML reader:

```
t.transform(new SAXSource(new EmployeeReader(),
   new InputSource(new FileInputStream(filename))), result);
```

This is an ingenious trick to convert non-XML legacy data into XML. Of course, most XSLT applications will already have XML input data, and you can simply invoke the transform method on a StreamSource:

```
t.transform(new StreamSource(file), result);
```

The transformation result is an object of a class that implements the Result interface. The Java API supplies three classes:

```
DOMResult
SAXResult
StreamResult
```

To store the result in a DOM tree, use a DocumentBuilder to generate a new document node and wrap it into a DOMResult:

```
Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));
```

To save the output in a file, use a StreamResult:

```
t.transform(source, new StreamResult(file));
```

Listing 3.12 contains the complete source code.

Listing 3.10 **v2ch03/transform/makehtml.xsl**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="html"/>

  <xsl:template match="/staff">
    <table border="1"><xsl:apply-templates/></table>
  </xsl:template>

  <xsl:template match="/staff/employee">
    <tr><xsl:apply-templates/></tr>
  </xsl:template>

  <xsl:template match="/staff/employee/name">
    <td><xsl:apply-templates/></td>
  </xsl:template>

  <xsl:template match="/staff/employee/salary">
    <td>$<xsl:apply-templates/></td>
  </xsl:template>

  <xsl:template match="/staff/employee/hiredate">
    <td><xsl:value-of select="@year"/>-<xsl:value-of
    select="@month"/>-<xsl:value-of select="@day"/></td>
  </xsl:template>

</xsl:stylesheet>
```

Listing 3.11 **v2ch03/transform/makeprop.xsl**

```
<?xml version="1.0"?>

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="text" omit-xml-declaration="yes"/>

  <xsl:template match="/staff/employee">
employee.<xsl:value-of select="position()"
/>.name=<xsl:value-of select="name/text()"/>
```

```
employee.<xsl:value-of select="position()"
/>.salary=<xsl:value-of select="salary/text()"/>
employee.<xsl:value-of select="position()"
/>.hiredate=<xsl:value-of select="hiredate/@year"
/>-<xsl:value-of select="hiredate/@month"
/>-<xsl:value-of select="hiredate/@day"/>
  </xsl:template>

</xsl:stylesheet>
```

Listing 3.12 **v2ch03/transform/TransformDemo.java**

```
package v2ch03.transform;

import module java.base;
import module java.xml;

import org.xml.sax.ContentHandler;
import v2ch07.util.Choices;

/**
 * This program demonstrates XSL transformations. It applies a transformation
 * employee records. The records are stored in the file employee.dat and turne
 * format.
 */
class TransformDemo {
   void main(String[] args) throws Exception {
      Path path;
      String choice = Choices.choose("HTML", "Properties").toLowerCase();
      if (choice.equals("html"))
         path = Path.of("v2ch03", "transform", "makehtml.xsl");
      else
         path = Path.of("v2ch03", "transform", "makeprop.xsl");
      try (InputStream styleIn = Files.newInputStream(path)) {
         var styleSource = new StreamSource(styleIn);

         Transformer t = TransformerFactory.newInstance().newTransformer(st
         t.setOutputProperty(OutputKeys.INDENT, "yes");
         t.setOutputProperty(OutputKeys.METHOD, "xml");
         t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");

         try (InputStream docIn = Files
               .newInputStream(Path.of("v2ch03", "transform", "employee.dat"))) {
```

```
            t.transform(new SAXSource(new EmployeeReader(), new InputSource
                  new StreamResult(System.out));
         }
      }
   }
}

/**
 * This class reads the flat file employee.dat and reports SAX parser events to a
 * was parsing an XML file.
 */
class EmployeeReader implements XMLReader {
   private ContentHandler handler;

   public void parse(InputSource source) throws IOException, SAXException {
      InputStream stream = source.getByteStream();
      var in = new BufferedReader(new InputStreamReader(stream));
      var atts = new AttributesImpl();

      if (handler == null) throw new SAXException("No content handler");

      handler.startDocument();
      handler.startElement("", "staff", "staff", atts);
      boolean done = false;
      while (!done) {
         String line = in.readLine();
         if (line == null)
            done = true;
         else {
            handler.startElement("", "employee", "employee", atts);
            var t = new StringTokenizer(line, "|");

            handler.startElement("", "name", "name", atts);
            String s = t.nextToken();
            handler.characters(s.toCharArray(), 0, s.length());
            handler.endElement("", "name", "name");

            handler.startElement("", "salary", "salary", atts);
            s = t.nextToken();
            handler.characters(s.toCharArray(), 0, s.length());
            handler.endElement("", "salary", "salary");

            atts.addAttribute("", "year", "year", "CDATA", t.nextToken());
            atts.addAttribute("", "month", "month", "CDATA", t.nextToken());
```

```
            atts.addAttribute("", "day", "day", "CDATA", t.nextToken());
            handler.startElement("", "hiredate", "hiredate", atts);
            handler.endElement("", "hiredate", "hiredate");
            atts.clear();

            handler.endElement("", "employee", "employee");
        }
    }

    handler.endElement("", "staff", "staff");
    handler.endDocument();
}

public void setContentHandler(ContentHandler newValue) {
    handler = newValue;
}

public ContentHandler getContentHandler() {
    return handler;
}

// the following methods are just do-nothing implementations
public void parse(String systemId) throws IOException, SAXException {
}

public void setErrorHandler(ErrorHandler handler) {
}

public ErrorHandler getErrorHandler() {
    return null;
}

public void setDTDHandler(DTDHandler handler) {
}

public DTDHandler getDTDHandler() {
    return null;
}

public void setEntityResolver(EntityResolver resolver) {
}

public EntityResolver getEntityResolver() {
    return null;
```

```
   }

   public void setProperty(String name, Object value) {
   }

   public Object getProperty(String name) {
      return null;
   }

   public void setFeature(String name, boolean value) {
   }

   public boolean getFeature(String name) {
      return false;
   }
}
```

javax.xml.transform.TransformerFactory1.4

- Transformer newTransformer(Source styleSheet)
  returns an instance of the Transformer class that reads a stylesheet from the given source.

javax.xml.transform.stream.StreamSource1.4

- StreamSource(File f)
- StreamSource(InputStream in)
- StreamSource(Reader in)
- StreamSource(String systemID)
  construct a stream source from a file, stream, reader, or system ID (usually a relative or absolute URL).

javax.xml.transform.sax.SAXSource1.4

- SAXSource(XMLReader reader, InputSource source)
  constructs a SAX source that obtains data from the given input source and uses the given reader to parse the input.

org.xml.sax.XMLReader 1.4

- void setContentHandler(ContentHandler handler)
  sets the handler that is notified of parse events as the input is parsed.
- void parse(InputSource source)
  parses the input from the given input source and sends parse events to the content handler.

javax.xml.transform.dom.DOMResult 1.4

- DOMResult(Node n)
  constructs a source from the given node. Usually, n is a new document node.

org.xml.sax.helpers.AttributesImpl 1.4

- void addAttribute(String uri, String lname, String qname, String type, String value)
  adds an attribute to this attribute collection. The lname parameter is the local name without prefix, and qname is the qualified name with prefix. The type parameter is one of "CDATA", "ID", "IDREF", "IDREFS", "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES", or "NOTATION".
- void clear()
  removes all attributes from this attribute collection.

This example concludes our discussion of XML support in the Java API. You should now have a good perspective on the major strengths of XML—in particular, its automated parsing and validation as well as its powerful transformation mechanism. Of course, all this technology is only going to work for you if you design your XML formats well. You need to make sure that the formats are rich enough to express all your business needs, that they are stable over time, and that your business partners are willing to accept your XML documents. Those issues can be

far more challenging than dealing with parsers, DTDs, or transformations.

In the next chapter, we will discuss network programming on the Java platform, starting with the basics of network sockets and moving on to higher-level protocols for e-mail and the World Wide Web.

Chapter 4 ▪ Networking

We begin this chapter by reviewing basic networking concepts, then move on to writing Java programs that connect to network services. We will show you how network clients and servers are implemented. Finally, you will see how to send e-mail from a Java program and how to harvest information from a web server.

4.1. Connecting to a Server

In the following sections, you will connect to a server, first by hand and with telnet, and then with a Java program.

4.1.1. Using Telnet

The telnet program is a great debugging tool for network programming. You should be able to launch it by typing telnet from a command shell.

> In Windows, you need to activate telnet. Go to the Control Panel, select Programs, click Turn Windows Features On or Off, and select the Telnet client checkbox. The Windows firewall also blocks quite a few network ports that we use in this chapter; you might need an administrator account to unblock them.

You may have used telnet to connect to a remote computer, but

you can use it to communicate with other services provided by Internet hosts as well. Here is an example of what you can do. Type

```
telnet time-a.nist.gov 13
```

As Figure 4.1 shows, you should get back a line like this:

```
57488 16-04-10 04:23:00 50 0 0 610.5 UTC(NIST) *
```



Figure 4.1: Output of the “time of day” service

What is going on? You have connected to the “time of day” service that most UNIX machines constantly run. The particular server that you connected to is operated by the National Institute of Standards and Technology and gives the measurement of a Cesium atomic clock. (Of course, the reported time is not completely accurate due to network delays.)

By convention, the “time of day” service is always attached to “port” number 13.

In network parlance, a port is not a physical device, but an abstraction facilitating communication between a server and a client (see Figure 4.2).

.

Figure 4.2: A client connecting to a server port

The server software is continuously running on the remote machine, waiting for any network traffic that wants to chat with port 13. When the operating system on the remote computer receives a network package that contains a request to connect to port number 13, it wakes up the listening server process and establishes the connection. The connection stays up until it is terminated by one of the parties.

When you began the telnet session with time-a.nist.gov at port 13, a piece of network software knew enough to convert the string "time-a.nist.gov" to its correct Internet Protocol (IP) address, 129.6.15.28. The telnet software then sent a connection request to that address, asking for a connection to port 13. Once the connection was established, the remote program sent back a line of data and closed the connection. In general, of course, clients and servers engage in a more extensive dialog before one or the other closes the connection.

Here is another experiment along the same lines—but a bit more interesting. Type

```
telnet horstmann.com 80
```

Then type very carefully the following:

```
GET /index.html HTTP/1.1
Host: horstmann.com
blank line
```

That is, hit the Enter key twice at the end.

Figure 4.3 shows the response. It should look eerily familiar—

you got a page of HTML-formatted text, namely Cay Horstmann's home page.



Figure 4.3: Using telnet to access an HTTP port

This is exactly the same process that your web browser goes through to get a web page. It uses HTTP to request web pages from servers. Of course, the browser displays the HTML code more nicely.

> The Host key/value pair is required when you connect to a web server that hosts multiple domains at the same IP address. You can omit it if the server hosts a single domain.

4.1.2. Connecting to a Server with Java

Our first network program in Listing 4.1 will do the same thing we did using telnet—connect to a port and print out what it finds.

Listing 4.1 **v2ch04/socket/SocketDemo.java**

```
package v2ch04.socket;

import module java.base;

/**
 * This program makes a socket connection to the atomic clock in Boulder, Colo
 * the time that the server sends.
 */
class SocketDemo {
   void main() throws Exception {
      try (var s = new Socket("time-a.nist.gov", 13);
            var in = new Scanner(s.getInputStream())) {
         while (in.hasNextLine()) {
            String line = in.nextLine();
            IO.println(line);
         }
      }
   }
}
```

The key statements of this simple program are these:

```
var s = new Socket("time-a.nist.gov", 13);
InputStream inStream = s.getInputStream();
```

The first line opens a socket, which is a network software abstraction that enables communication out of and into this program. We pass the remote address and the port number to the socket constructor. If the connection fails, an UnknownHostException is thrown, which is a subclass of IOException.

The getInputStream method in java.net.Socket returns an InputStream object that you can use just like any other stream. The method throws an IOException if it is not possible to read from the socket. Once you have grabbed the stream, this program simply prints each input line to standard output. This process continues until the stream is finished and the server disconnects.

This program works only with very simple servers, such as a "time of day" service. In more complex networking programs, the client sends request data to the server, and the server might not immediately disconnect at the end of a response. You will see how to implement that behavior in several examples throughout this chapter.

The Socket class is pleasant and easy to use because the Java library hides the complexities of establishing a networking connection and sending data across it. The java.net package essentially gives you the same programming interface you would use to work with a file.

In this book, we cover only the Transmission Control Protocol (TCP). The Java platform also supports the User Datagram Protocol (UDP), which can be used to send packets (also called datagrams) with much less overhead than TCP. The drawback is that packets need not be delivered in sequential order to the receiving application and can even be dropped altogether. It is up to the recipient to put the packets in order and to request retransmission of missing packets. UDP is well suited for applications in which missing packets can be tolerated—for example, for audio or video streams or continuous measurements.

---

java.net.Socket **1.0**

- Socket(String host, int port)
  constructs a socket to connect to the given host and port.
- InputStream getInputStream()
- OutputStream getOutputStream()
  get the stream to read data from the socket or write data to the socket.

4.1.3. Socket Timeouts

Reading from a socket blocks until data are available. If the host is unreachable, your application waits for a long time and you are at the mercy of the underlying operating system to eventually time out.

You can decide what timeout value is reasonable for your particular application. Then, call the setSoTimeout method to set a timeout value (in milliseconds).

```
var s = new Socket(. . .);
s.setSoTimeout(10000); // time out after 10 seconds
```

If the timeout value has been set for a socket, all subsequent read operations throw a SocketTimeoutException when the timeout has been reached before the operation has completed its work. You can catch that exception and react to the timeout.

```
try {
   InputStream in = s.getInputStream(); // read from in
   . . .
}
catch (SocketTimeoutException e) {
   react to timeout
}
```

There is no timeout for write operations.

There is one additional timeout issue that you need to address. The constructor

```
Socket(String host, int port)
```

can block indefinitely until an initial connection to the host is established.

You can overcome this problem by first constructing an unconnected socket and then connecting it with a timeout:

```
var s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

See Section 4.2.4 for how to allow users to interrupt a socket

connection at any time.

java.net.Socket1.0

- Socket()1.1
  creates a socket that has not yet been connected.
- void connect(SocketAddress address)1.4
  connects this socket to the given address.
- void connect(SocketAddress address, int timeoutInMilliseconds)1.4
  connects this socket to the given address, or throws a SocketTimeoutException if the time interval expired.
- void setSoTimeout(int timeoutInMilliseconds)1.1
  sets the blocking time for read requests on this socket. If the timeout is reached when reading, a SocketTimeoutException is thrown.
- boolean isConnected()1.4
- boolean isClosed()1.4
  return true if the socket is connected or closed.

4.1.4. Internet Addresses

Usually, you don't have to worry too much about Internet addresses—the numerical host addresses that consist of 4 bytes (or, with IPv6, 16 bytes) such as 129.6.15.28. However, you can use the InetAddress class if you need to convert between host names and Internet addresses.

The java.net package supports IPv6 Internet addresses, provided the host operating system does.

The static getByName method returns an InetAddress object of a host. For example,

```
InetAddress address = InetAddress.getByName("time-a.nist.gov");
```

returns an InetAddress object that encapsulates the sequence of four bytes 129.6.15.28. You can access the bytes with the

getAddress method.

```
byte[] addressBytes = address.getAddress();
```

Some host names with a lot of traffic correspond to multiple Internet addresses, to facilitate load balancing. For example, at the time of this writing, the host name google.com corresponds to twelve different Internet addresses. One of them is picked at random when the host is accessed. You can get all hosts with the getAllByName method.

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

Finally, you sometimes need the address of the local host. If you simply ask for the address of localhost, you always get the local loopback address 127.0.0.1, which cannot be used by others to connect to your computer. Instead, use the static getLocalHost method to get the address of your local host.

```
InetAddress address = InetAddress.getLocalHost();
```

Listing 4.2 is a simple program that prints the Internet address of your local host if you do not specify any command-line arguments, or all Internet addresses of another host if you specify the host name on the command line, such as

```
java v2ch04.inetAddress.InetAddressDemo www.horstmann.com
```

Listing 4.2 **v2ch04/inetAddress/InetAddressDemo.java**

```
package v2ch04.inetAddress;

import module java.base;

/**
 * This program demonstrates the InetAddress class. Supply a host name as co
 * argument, or run without command-line arguments to see the address of th
 */
class InetAddressDemo {
   void main(String[] args) throws Exception {
      if (args.length > 0) {
```

```
            String host = args[0];
            InetAddress[] addresses = InetAddress.getAllByName(host);
            for (InetAddress a : addresses)
               System.out.println(a);
         }
         else {
            InetAddress localHostAddress = InetAddress.getLocalHost();
            System.out.println(localHostAddress);
         }
      }
}
```

java.net.InetAddress1.0

- static InetAddress getByName(String host)
- static InetAddress[] getAllByName(String host)
  construct an InetAddress, or an array of all Internet addresses, for the given host name.
- static InetAddress getLocalHost()
  constructs an InetAddress for the local host.
- byte[] getAddress()
  returns an array of bytes that contains the numerical address.
- String getHostAddress()
  returns a string with decimal numbers, separated by periods, for example "129.6.15.28".
- String getHostName()
  returns the host name.

4.2. Implementing Servers

Now that we have implemented a basic network client that receives data from the Internet, let's program a simple server that can send information to clients.

4.2.1. Server Sockets

A server program, when started, waits for a client to attach to its port. For our example program, we chose port number 8189,

which is not used by any of the standard services. The ServerSocket class establishes a socket. In our case, the command

```
var s = new ServerSocket(8189);
```

establishes a server that monitors port 8189. The command

```
Socket incoming = s.accept();
```

tells the program to wait indefinitely until a client connects to that port. Once someone connects to this port by sending the correct request over the network, this method returns a Socket object that represents the connection that was made. You can use this object to get input and output streams, as is shown in the following code:

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

Everything that the server sends to the server output stream becomes the input of the client program, and all the output from the client program ends up in the server input stream.

In all the examples in this chapter, we transmit text through sockets. We therefore turn the streams into scanners and writers.

```
var in = new Scanner(inStream);
var out = new PrintWriter(new OutputStreamWriter(outStream), tru
```

Let’s send the client a greeting:

```
out.println("Hello! Enter BYE to exit.");
```

When you use telnet to connect to this server program at port 8189, you will see this greeting on the terminal screen.

In this simple server, we just read the client’s input, a line at a time, and echo it. This demonstrates that the program receives the input. An actual server would obviously compute and return an answer depending on the input.

```
   String line = in.nextLine();
   out.println("Echo: " + line);
   if (line.strip().equals("BYE")) done = true;
```

In the end, we close the incoming socket.

```
   incoming.close();
```

That is all there is to it. Every server program, such as an HTTP web server, continues performing this loop:

1. It receives a command from the client ("get me this information") through an incoming data stream.
2. It decodes the client command.
3. It gathers the information that the client requested.
4. It sends the information to the client through the outgoing data stream.

Listing 4.3 is the complete program.

Listing 4.3 **v2ch04/server/EchoServer.java**

```
package v2ch04.server;

import module java.base;

/**
 * This program implements a simple server that listens to port 8189 or a port
 * argument, and echoes back all client input.
 */
class EchoServer {
   void main(String[] args) throws Exception {
      int port = args.length >= 1 ? Integer.parseInt(args[0]) : 8189;

      try (var s = new ServerSocket(port); // establish server socket
            Socket incoming = s.accept()) { // wait for client connection
         serve(incoming);
      }
   }

   void serve(Socket incoming) throws IOException {
```

```
        try (var in = new Scanner(incoming.getInputStream());
             var out = new PrintWriter(incoming.getOutputStream(), true /* autoF
            out.println("Hello! Enter BYE to exit.");

            // echo client input
            boolean done = false;
            while (!done && in.hasNextLine()) {
               String line = in.nextLine();
               out.println("Echo: " + line);
               if (line.strip().equals("BYE")) done = true;
            }
        }
    }
}
```

To try it out, compile and run the program. Then use telnet to connect to the server localhost (or IP address 127.0.0.1) and port 8189.

If you are connected directly to the Internet, anyone in the world can access your echo server, provided they know your IP address and the magic port number.

When you connect to the port, you will see the message shown in Figure 4.4:

```
Hello! Enter BYE to exit.
```

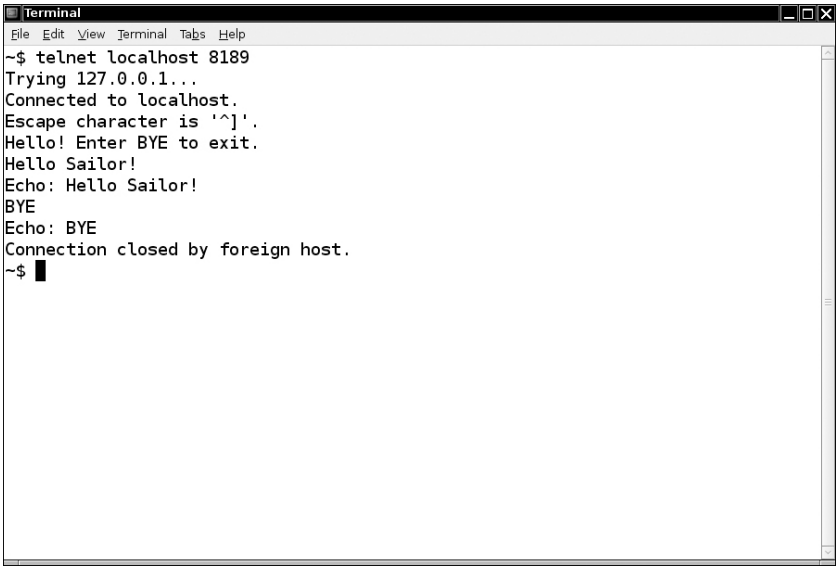


Figure 4.4: Accessing an echo server

Type anything and watch the input echo on your screen. Type BYE (all uppercase letters) to disconnect. The server program will terminate as well.

java.net.ServerSocket1.0

- ServerSocket(int port)
  creates a server socket that monitors a port.
- Socket accept()
  waits for a connection. This method blocks (i.e., idles) the current thread until the connection is made. The method returns a Socket object through which the program can communicate with the connecting client.
- void close()
  closes the server socket.

4.2.2. Serving Multiple Clients

There is one problem with the simple server in the preceding

example. Suppose we want to allow multiple clients to connect to our server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet might want to use it at the same time. Without support for multiple connections, any one client can monopolize the service by connecting to it for a long time. We can do much better through the magic of threads.

Every time we know the program has established a new socket connection—that is, every time the call to accept() returns a socket—we will launch a new thread to take care of the connection between the server and that client. The main program will just go back and wait for the next connection. It keeps going forever, or until the server process is killed:

```
ExecutorService service = Executors.newVirtualThreadPerTaskExec
while (true) {
   Socket incoming = s.accept();
   service.submit(() -> serve(incoming));
}
```

The serve method contains the communication loop with the client:

```
public void serve(Socket incoming) {
   try (var in = new Scanner(incoming.getInputStream());
         var out = new PrintWriter(incoming.getOutputStream(),
         true /* autoFlush */)) {
      Process input and send response
   }
   catch (IOException e) {
     Handle exception
   }
}
```

When each connection starts a new thread, multiple clients can connect to the server at the same time. You can easily check this out.

1. Compile and run the server program (Listing 4.4).
2. Open several telnet windows as we have in Figure 4.5.

3. Switch between windows and type commands. Note that you can communicate through all of them simultaneously.
4. When done, switch to the window from which you launched the server program and press Ctrl+C to kill it.

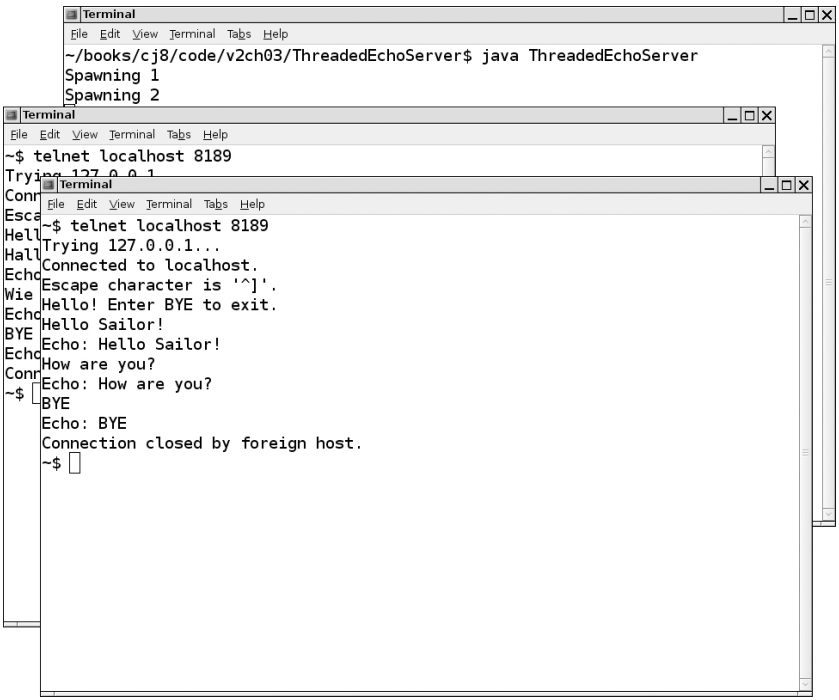


Figure 4.5: Several telnet windows communicating simultaneously

---

Listing 4.4 **v2ch04/threaded/EchoServer.java**

```
package v2ch04.threaded;

import module java.base;

/**
 * This program implements a multithreaded server that listens to port 8189 a
 * client input.
```

```
 */
class EchoServer {
   void main() throws Exception {
      try (var s = new ServerSocket(8189)) {
         int i = 1;
         ExecutorService service = Executors.newVirtualThreadPerTaskExecuto

         while (true) {
            Socket incoming = s.accept();
            IO.println("Spawning " + i);
            service.submit(() -> serve(incoming));
            i++;
         }
      }
   }

   void serve(Socket incoming) {
      try (var in = new Scanner(incoming.getInputStream());
           var out = new PrintWriter(incoming.getOutputStream(), true /* autoF
         out.println("Hello! Enter BYE to exit.");

         // echo client input
         boolean done = false;
         while (!done && in.hasNextLine()) {
            String line = in.nextLine();
            out.println("Echo: " + line);
            if (line.strip().equals("BYE")) done = true;
         }
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

4.2.3. Half-Close

The half-close allows one end of a socket connection to terminate its output while still receiving data from the other end.

Here is a typical situation. Suppose you transmit data to the server but you don’t know at the outset how much data you

have. With a file, you'd just close the file at the end of the data. However, if you close a socket, you immediately disconnect from the server and cannot read the response.

The half-close solves this problem. You can close the output stream of a socket, thereby indicating to the server the end of the requested data, but keep the input stream open.

The client side looks like this:

```
try (var socket = new Socket(host, port)) {
   var in = new Scanner(socket.getInputStream());
   var writer = new PrintWriter(socket.getOutputStream());
   // send request data
   writer.print(. . .);
   writer.flush();
   socket.shutdownOutput();
   // now socket is half-closed
   // read response data
   while (in.hasNextLine()) {
      String line = in.nextLine();
      . . .
   }
}
```

The server side simply reads input until the end of the input stream is reached. Then it sends the response.

Of course, this protocol is only useful for one-shot services such as HTTP where the client connects, issues a request, catches the response, and then disconnects.

java.net.Socket1.0

- void shutdownInput()1.3
- void shutdownOutput()1.3
  set the input or output stream to "end of stream."
- boolean isInputShutdown()1.4
- boolean isOutputShutdown()1.4
  return true if input or output has been shut down.

4.2.4. Interruptible Sockets

When you connect to a socket, the current thread blocks until the connection has been established or a timeout has elapsed. Similarly, when you read data through a socket, the current thread blocks until the operation is successful or has timed out. (There is no timeout for writing data.)

In interactive applications, you would like to give users an option to simply cancel a socket connection that does not appear to produce results. However, if a platform thread blocks on an unresponsive socket, you cannot unblock it by calling interrupt.

> When using virtual threads, socket operations are interruptible, and you do not need the technique discussed in this section.

To interrupt a socket operation, use a SocketChannel, a feature of the java.nio package. Open the SocketChannel like this:

```
SocketChannel channel = SocketChannel.open(new InetSocketAddre
```

A channel does not have associated streams. Instead, it has read and write methods that make use of Buffer objects. (See Chapter 2 for more information about NIO buffers.) These methods are declared in the interfaces ReadableByteChannel and WritableByteChannel.

If you don't want to deal with buffers, you can use the Scanner class to read from a SocketChannel because Scanner has a constructor with a ReadableByteChannel parameter:

```
var in = new Scanner(channel);
```

To turn a channel into an output stream, use the static Channels.newOutputStream method.

```
OutputStream outStream = Channels.newOutputStream(channel);
```

That's all you need to do. Whenever a thread is interrupted during an open, read, or write operation, the operation does not block, but is terminated with an exception.

The program in Listing 4.5 contrasts interruptible and blocking sockets. A server sends numbers and pretends to be stuck after the tenth number. You are asked whether you want to use a socket or a channel. You can also choose between platform and virtual threads. A client thread is started that connects to the server and prints the output. Hit Enter to interrupt the client thread. If you do so within the first ten numbers, you can interrupt the thread.

However, if you use a socket and a platform thread, interruption after the first ten numbers does not work. The blocking thread keeps blocking until the server finally closes the connection. The remedy is to use a channel or virtual threads.

Listing 4.5 **v2ch04/interruptible/InterruptibleSocketDemo.java**

```
package v2ch04.interruptible;

import module java.base;

/**
 * This program shows how to interrupt a socket channel.
 */
class InterruptibleSocketDemo {
   void main() {
      Thread.ofPlatform().start(new TestServer());
      boolean socket = IO.readln("Socket or channel? (S/C) ").equalsIgnoreCase("
      boolean platform = IO.readln("Platform or virtual? (P/V) ").equalsIgnoreCa
      Runnable client = socket ? this::useSocket : this::useChannel;
      Thread clientThread = platform ? Thread.ofPlatform().start(client)
            : Thread.ofVirtual().start(client);

      IO.readln("Hit Enter to interrupt client");
      clientThread.interrupt();
```

```
}

/**
 * Connects to the test server, using a channel.
 */
void useChannel() {
   IO.println("Channel:\n");
   try (SocketChannel channel = SocketChannel
         .open(new InetSocketAddress("localhost", 8189));
         Scanner in = new Scanner(channel)) {
      while (!Thread.currentThread().isInterrupted()) {
         IO.print("Reading ");
         if (in.hasNextLine()) {
            String line = in.nextLine();
            IO.println(line);
         }
      }
   }
   catch (IOException e) {
      e.printStackTrace();
   }
   finally {
      IO.println("Channel closed\n");
   }
}

/**
 * Connects to the test server, using a socket.
 */
void useSocket() {
   IO.println("Socket:\n");
   try (var sock = new Socket("localhost", 8189);
         Scanner in = new Scanner(sock.getInputStream())) {
      while (!Thread.currentThread().isInterrupted()) {
         IO.print("Reading ");
         if (in.hasNextLine()) {
            String line = in.nextLine();
            IO.println(line);
         }
      }
   }
   catch (IOException e) {
      e.printStackTrace();
   }
```

```
      finally {
         IO.println("Socket closed\n");
      }
   }

   /**
    * A server that listens to port 8189 and sends numbers to the client, simulat
    * server after 10 numbers.
    */
   static class TestServer implements Runnable {
      public void run() {
         try (var s = new ServerSocket(8189); Socket incoming = s.accept()) {
            serve(incoming);
         }
         catch (Exception e) {
            e.printStackTrace();
            ;
         }
         finally {
            IO.println("Closing connection\n");
         }
      }

      private void serve(Socket incoming) throws IOException, InterruptedExce
         int counter = 0;
         try (var out = new PrintWriter(incoming.getOutputStream(), true /* auto
            while (counter < 100) {
               counter++;
               if (counter <= 10) out.println(counter);
               Thread.sleep(100);
            }
         }
      }
   }
}
```

java.net.InetSocketAddress 1.4

- InetSocketAddress(String hostname, int port)
  constructs an address object with the given host and port, resolving the host name during construction. If the host name cannot be resolved, the address object's unresolved property is set to true.
- boolean isUnresolved()

returns true if this address object could not be resolved.

java.nio.channels.SocketChannel1.4

- static SocketChannel open(SocketAddress address)
  opens a socket channel and connects it to a remote address.

java.nio.channels.Channels1.4

- static InputStream newInputStream(ReadableByteChannel channel)
  constructs an input stream that reads from the given channel.
- static OutputStream newOutputStream(WritableByteChannel channel)
  constructs an output stream that writes to the given channel.

4.2.5. Secure Socket Communication

When you implement clients and servers across the Internet and you exchange confidential information, you do not want to use plain text traffic. Instead, you should use a secure communication mechanism. There are three aspects to secure communication:

- The communication is encrypted, so it cannot be read by third parties.
- The identity of the server is authenticated with a digital signature that the client verifies. Optionally, the client can also provide a digital signature, but this is not common for web applications.
- Transmitted data are digitally signed by the sender and verified by the receiver, guaranteeing that no third party modified or substituted the data.

The TLS (Transport Layer Security) protocol is the most common mechanism for secure communication. It is a successor to the SSL (Secure Socket Layer) protocol.

In Java, you use classes from the javax.net.ssl package to establish secure connections. Here is how to construct a socket for a secure connection:

```
SocketFactory factory = SSLSocketFactory.getDefault();
Socket s = factory.createSocket(host, port);
```

For secure server sockets, you also use a factory:

```
ServerSocketFactory factory = SSLServerSocketFactory.getDefault();
ServerSocket s = factory.createServerSocket(port);
```

If you need to configure the secure connection parameters of the client or server socket, cast the object returned by the factory to a SSLSocket or SSLServerSocket:

```
((SSLServerSocket) s).setNeedClientAuth(true);
```

The client and server encrypt the traffic, using a shared encryption key. However, they need to know that they are really talking to each other and not to a "man in the middle." The server presents a certificate to vouch for its authenticity. The certificate needs to be signed by a certificate authority known to the client. The JDK has a built-in list of known certificate authorities. You would normally obtain a server certificate from one of them and install it in the server. However, for testing, we will generate our own certificates.

See Chapter 9 for a detailed description of certificates and the keytool program used to manage them. Here are the commands to generate the certificates and place them in a "key store" for the server and client.

First, generate the server certificate:

```
keytool -genkeypair -alias servercert -keyalg RSA -keysize 2048 -sigal
   -keystore server.jks -storepass secret -ext san=ip:127.0.0.1,dns:loca
```

You will be asked a number of questions about the organization in charge of the server. For this demonstration, you can leave them all to the default value of “unknown”.

Next, export the certificate and import it into the client store.

```
keytool -exportcert -keystore server.jks -storepass secret -alias server
   -file servercert.pem
keytool -import -trustcacerts -file servercert.pem -keystore client.jks
   -storepass secret -keypass secret
```

When importing the certificate, which is not signed by one of the known certificate authorities, you will be asked whether you trust it. Answer with yes.

Now start the server. You need to specify the keystore containing the server certificate.

```
java -Djavax.net.ssl.keyStore=server.jks -Djavax.net.ssl.keyStorePassw
   v2ch04.ssl.EchoServer
```

Then run the client, specifying the keystore containing the certificate that the client trusts. This would not be necessary if the certificate was signed by a certificate authority that is known to the JDK.

```
java -Djavax.net.ssl.trustStore=client.jks -Djavax.net.ssl.trustStorePas
   v2ch04.ssl.EchoClient
```

If there are connection problems, try running the client with the -Djavax.net.debug=ssl command-line option.

Listing 4.6 **v2ch04/ssl/EchoClient.java**

```
package v2ch04.ssl;

import module java.base;

/**
```

```
 * This program connects to a server via SSL and echoes the server responses.
 */
class EchoClient {
   void main(String[] args) throws Exception {
      int port = args.length >= 1 ? Integer.parseInt(args[0]) : 8189;
      SocketFactory factory = SSLSocketFactory.getDefault();
      String message = """
Hello
World
BYE
""";
      try (Socket s = factory.createSocket(InetAddress.getLocalHost(), port);
            OutputStream os = s.getOutputStream();
            var in = new Scanner(s.getInputStream())) {
         os.write(message.getBytes());
         os.flush();
         boolean done = false;
         while (!done) {
            String line = in.nextLine();
            System.out.println(line);
            if (line.equals("Echo: BYE")) done = true;
         }
      }
   }
}
```

Listing 4.7 **v2ch04/ssl/EchoServer.java**

```
package v2ch04.ssl;

import module java.base;

/**
 * This program implements a simple server that listens to port 8189
 * or a port given as argument, and echoes back all client input.
 */
class EchoServer {
   void main(String[] args) throws Exception {
      int port = args.length >= 1 ? Integer.parseInt(args[0]) : 8189;
      // establish server socket
      ServerSocketFactory factory = SSLServerSocketFactory.getDefault();
      try (ServerSocket s = factory.createServerSocket(port)) {
         ExecutorService service = Executors.newVirtualThreadPerTaskExecutor
```

```
            while (true) {
               // wait for client connection
               Socket incoming = s.accept();
               service.submit(() -> serve(incoming));
            }
         }
      }

      void serve(Socket incoming) {
         try (var in = new Scanner(incoming.getInputStream());
              var out = new PrintWriter(incoming.getOutputStream(),
                    true /* autoFlush */)) {
            out.println("Hello! Enter BYE to exit.");

            // echo client input
            boolean done = false;
            while (!done && in.hasNextLine()) {
               String line = in.nextLine();
               out.println("Echo: " + line);
               if (line.strip().equals("BYE"))
                  done = true;
            }
         }
         catch (IOException e) {
            e.printStackTrace();
         }
      }
   }
```

javax.net.ServerSocketFactory 1.4

- static ServerSocket createServerSocket(int port)
  yields a server socket for the given port.

javax.net.ssl.SSLServerSocketFactory 1.4

- static ServerSocketFactory getDefault()
  returns a factory that produces SSLServerSocket instances.

javax.net.ssl.SSLSocketFactory1.4

- static SocketFactory getDefault()
  returns a factory that produces SSLSocket instances.

javax.net.SocketFactory1.4

- static Socket createSocket(InetAddress host, int port)
- static Socket createSocket(String host, int port)
  yield a socket for connecting to the given host and port.

4.3. Getting Web Data

To access web servers in a Java program, you will want to work at a higher level than socket connections and HTTP requests. In the following sections, we discuss the classes that the Java library provides for this purpose.

4.3.1. URLs and URIs

The java.net package makes a useful distinction between URLs (uniform resource locators) and URIs (uniform resource identifiers).

A URI is a purely syntactical construct that contains the various parts of the string specifying a web resource. A URL is a special kind of URI, namely one with sufficient information to locate a resource. Other URIs, such as

```
mailto:cay@horstmann.com
```

are not locators—there is no data to locate from this identifier. Such a URI is called a URN (uniform resource name).

In the Java library, the URI class has no methods for accessing the resource that the identifier specifies—its sole purpose is parsing. In contrast, the URL class can open a stream to the

resource. For that reason, the URL class only works with schemes that the Java library knows how to handle, such as http:, https:, ftp:, the local file system (file:), and JAR files (jar:).

You can construct a URL object from a URI:

```
var url = new URI(urlString).toURL();
```

> The URL class has constructors that were deprecated in Java 20. They have various technical issues that make it possible to construct invalid URLs. Instead, first construct a URI, and then obtain the URL, as shown above.

If you simply want to fetch the contents of the resource, use the openStream method of the URL class. This method yields an InputStream object. Use it in the usual way—for example, to construct a Scanner:

```
InputStream inStream = url.openStream();
var in = new Scanner(inStream);
```

In the next section, you will see how to use the URLConnection class for more control over the connection.

Let us return to URIs. To see why parsing is not trivial, consider how complex URIs can be. For example,

```
https://google.com?q=Beach+Chalet
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

The URI specification gives the rules for the makeup of these identifiers. A URI has the syntax

```
[scheme:]schemeSpecificPart[#fragment]
```

Here, the brackets [ ] enclose optional parts, and the : and # are included literally in the identifier.

If the *scheme*: part is present, the URI is called absolute. Otherwise, it is called relative.

An absolute URI is opaque if the *schemeSpecificPart* does not begin with a / such as

mailto:cay@horstmann.com

All absolute nonopaque URIs and all relative URIs are hierarchical. Examples are

http://horstmann.com/index.html
../../java/net/Socket.html#Socket()

The *schemeSpecificPart* of a hierarchical URI has the structure

[//*authority*][*path*][?*query*]

where, again, the brackets [ ] enclose optional parts.

For server-based URIs, the *authority* part has the form

[*user-info*@]*host*[:*port*]

The *port* must be an integer.

RFC 2396, which standardizes URIs, also supports a registry-based mechanism in which the *authority* has a different format, but this is not in common use.

One of the purposes of the URI class is to parse an identifier and break it up into its components. You can retrieve them with the methods

getScheme
getSchemeSpecificPart
getAuthority
getUserInfo
getHost
getPort
getPath
getQuery
getFragment

The other purpose of the URI class is the handling of absolute

and relative identifiers. If you have an absolute URI such as

https://docs.mycompany.com/api/java/net/ServerSocket.html

and a relative URI such as

../../java/net/Socket.html#Socket()

then you can combine the two into an absolute URI.

https://docs.mycompany.com/api/java/net/Socket.html#Socket()

This process is called resolving a relative URL.

The opposite process is called relativization. For example, suppose you have a base URI

https://docs.mycompany.com/api

and a URI

https://docs.mycompany.com/api/java/lang/String.html

Then the relativized URI is

java/lang/String.html

The URI class supports both of these operations:

```
relative = base.relativize(combined);
combined = base.resolve(relative);
```

4.3.2. Using a URLConnection to Retrieve Information

If you want additional information about a web resource, you should use the URLConnection class, which gives you much more control than the basic URL class.

When working with a URLConnection object, you must carefully schedule your steps.

1. Call the openConnection method of the URL class to obtain the URLConnection object:

   ```
   URLConnection connection = url.openConnection();
   ```

2. Set any request properties, using the methods

   ```
   setDoInput
   setDoOutput
   setIfModifiedSince
   setUseCaches
   setAllowUserInteraction
   setRequestProperty
   setConnectTimeout
   setReadTimeout
   ```

   We discuss these methods later in this section and in the API notes.
3. Connect to the remote resource by calling the connect method:

   ```
   connection.connect();
   ```

   Besides making a socket connection to the server, this method also queries the server for header information.
4. After connecting to the server, you can query the header information. Two methods, getHeaderFieldKey and getHeaderField, enumerate all fields of the header. The method getHeaderFields gets a standard Map object containing the header fields. For your convenience, the following methods query standard fields:

   ```
   getContentType
   getContentLength
   getContentEncoding
   getDate
   getExpiration
   getLastModified
   ```

5. Finally, you can access the resource data. Use the getInputStream method to obtain an input stream for

reading the information. (This is the same input stream that the openStream method of the URL class returns.) The other method, getContent, isn't very useful in practice. The objects that are returned by standard content types such as text/plain and image/gif require classes in the com.sun hierarchy for processing. You could register your own content handlers, but we do not discuss this technique in our book.

Some programmers form a wrong mental image when using the URLConnection class, thinking that the getInputStream and getOutputStream methods are similar to those of the Socket class. But that isn't quite true. The URLConnection class does quite a bit of magic behind the scenes—in particular, the handling of request and response headers. For that reason, it is important that you follow the setup steps for the connection.

Let us now look at some of the URLConnection methods in detail. Several methods set properties of the connection before connecting to the server. The most important ones are setDoInput and setDoOutput. By default, the connection yields an input stream for reading from the server but no output stream for writing. If you want an output stream (for example, for posting data to a web server), you need to call

```
connection.setDoOutput(true);
```

Next, you may want to set some of the request headers. The request headers are sent together with the request command to the server. Here is an example:

```
GET www.server.com/index.html HTTP/1.0
Referer: https://www.somewhere.com/links.html
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Cookie: orangemilano=192218887821987

The setIfModifiedSince method tells the connection that you are only interested in data modified since a certain date.

Finally, you can use the catch-all setRequestProperty method to set any name/value pair that is meaningful for the particular protocol. For the format of the HTTP request headers, see RFC 2616. Some of these properties are not well documented and are passed around by word of mouth from one programmer to the next. For example, if you want to access a password-protected web page, you must do the following:

1. Concatenate the username, a colon, and the password.

   ```
   String input = username + ":" + password;
   ```

2. Compute the Base64 encoding of the resulting string. (The Base64 encoding encodes a sequence of bytes into a sequence of printable ASCII characters.)

   ```
   Base64.Encoder encoder = Base64.getEncoder();
   String encoding = encoder.encodeToString(input.getBytes());
   ```

3. Call the setRequestProperty method with a name of "Authorization" and the value "Basic " + encoding.

   ```
   connection.setRequestProperty("Authorization", "Basic " + enco
   ```

You just saw how to access a password-protected web page. To access a password-protected file by FTP, use an entirely different method: Construct a URL of the form

ftp://username:password@ftp.yourserver.com/pub/file.txt

Once you call the connect method, you can query the response header information. First, let's see how to enumerate all response header fields. The implementors of this class felt a need to express their individuality by introducing yet another iteration protocol. The call

```
String key = connection.getHeaderFieldKey(n);
```

gets the nth key from the response header, where n starts from 1! It returns null if n is less than 1 or greater than the total number of header fields. There is no method to return the number of fields; you simply keep calling getHeaderFieldKey until you get null. Similarly, the call

```
String value = connection.getHeaderField(n);
```

returns the nth value.

The method getHeaderFields returns a Map of response header fields.

```
Map<String,List<String>> headerFields = connection.getHeaderField
```

Here is a set of response header fields from a typical HTTP request:

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

> Even though connection.getHeaderFieldKey(0) returns null, you can call connection.getHeaderField(0) to get the response status line (such as "HTTP/1.1 200 OK"). You also get it when you call headerFields.get(null).

As a convenience, six methods query the values of the most common header types and convert them to numeric types when appropriate. Table 4.1 shows these convenience methods. The methods with return type long return the number of seconds since January 1, 1970 GMT.

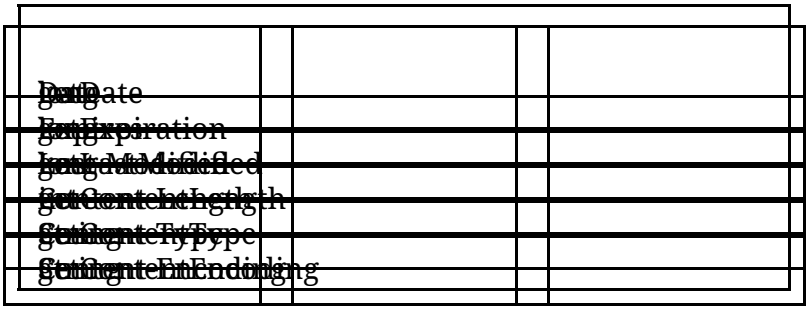


Table 4.1: Convenience Methods for Response Header Values

The program in Listing 4.8 lets you experiment with URL connections. Supply a URL and an optional username and password on the command line when running the program, for example:

```
java urlConnection.URLConnectionDemo https://horstmann.com/cor
```

The program prints

- All keys and values of the header
- The return values of the six convenience methods in Table 4.1
- The first ten lines of the requested resource

Listing 4.8 **v2ch04/urlConnection/URLConnectionDemo.java**

```
package v2ch04.urlConnection;

import module java.base;

/**
 * This program connects to an URL and displays the response header data and
 * of the requested data.
 *
 * Supply the URL and an optional username and password (for HTTP basic au
 * command line.
 */
class URLConnectionDemo {
   void main(String[] args) throws Exception {
      String urlName = args.length > 0 ? args[0] : "http://horstmann.com";
```

```
var url = new URI(urlName).toURL();
URLConnection connection = url.openConnection();

// set username, password if specified on command line

if (args.length > 2) {
   String username = args[1];
   String password = args[2];
   String input = username + ":" + password;
   Base64.Encoder encoder = Base64.getEncoder();
   String encoding = encoder.encodeToString(input.getBytes());
   connection.setRequestProperty("Authorization", "Basic " + encoding);
}

connection.connect();

// print header fields

Map<String, List<String>> headers = connection.getHeaderFields();
for (Map.Entry<String, List<String>> entry : headers.entrySet()) {
   String key = entry.getKey();
   for (String value : entry.getValue())
      IO.println(key + ": " + value);
}

// print convenience functions

IO.println("----------");
IO.println("getContentType: " + connection.getContentType());
IO.println("getContentLength: " + connection.getContentLength());
IO.println("getContentEncoding: " + connection.getContentEncoding());
IO.println("getDate: " + connection.getDate());
IO.println("getExpiration: " + connection.getExpiration());
IO.println("getLastModifed: " + connection.getLastModified());
IO.println("----------");

String encoding = connection.getContentEncoding();
if (encoding == null) encoding = "UTF-8";
try (var in = new Scanner(connection.getInputStream(), encoding)) {
   // print first ten lines of contents

   for (int n = 1; in.hasNextLine() && n <= 10; n++)
      IO.println(in.nextLine());
   if (in.hasNextLine()) IO.println(". . .");
```

```
    }
  }
}
```

java.net.URL1.0

- InputStream openStream()
  opens an input stream for reading the resource data.
- URLConnection openConnection()
  returns a URLConnection object that manages the connection to the resource.

java.net.URLConnection1.0

- void setDoInput(boolean doInput)
- boolean getDoInput()
  If doInput is true, the user can receive input from this URLConnection.
- void setDoOutput(boolean doOutput)
- boolean getDoOutput()
  If doOutput is true, the user can send output to this URLConnection.
- void setIfModifiedSince(long time)
- long getIfModifiedSince()
  The ifModifiedSince property configures this URLConnection to fetch only data modified since a given time. The time is given in seconds since midnight, GMT, January 1, 1970.
- void setConnectTimeout(int timeout)5.0
- int getConnectTimeout()5.0
  set or get the timeout for the connection (in milliseconds). If the timeout has elapsed before a connection was established, the connect method of the associated input stream throws a SocketTimeoutException.
- void setReadTimeout(int timeout)5.0
- int getReadTimeout()5.0
  set or get the timeout for reading data (in milliseconds). If the timeout has elapsed before a read operation was successful, the read method throws a

SocketTimeoutException.

- void setRequestProperty(String key, String value)
  sets a request header field.
- Map<String,List<String>> getRequestProperties()1.4
  returns a map of request properties. All values for the same key are placed in a list.
- void connect()
  connects to the remote resource and retrieves the response header information.
- Map<String,List<String>> getHeaderFields()1.4
  returns a map of response headers. All values for the same key are placed in a list.
- String getHeaderFieldKey(int n)
  gets the key for the nth response header field, or null if n is ≤0 or greater than the number of response header fields.
- String getHeaderField(int n)
  gets value of the nth response header field, or the response status line if n is 0 or null if n is <0 or greater than the number of response header fields.
- int getContentLength()
  gets the content length if available, or -1 if unknown.
- String getContentType()
  gets the content type, such as text/plain or image/gif.
- String getContentEncoding()
  gets the content encoding, such as gzip. This value is not commonly used, because the default identity encoding is not supposed to be specified with a Content-Encoding header.
- long getDate()
- long getExpiration()
- long getLastModifed()
  get the date of creation, expiration, and last modification of the resource. The dates are specified as seconds since midnight, GMT, January 1, 1970.
- InputStream getInputStream()
- OutputStream getOutputStream()
  return a stream for reading from the resource or writing to the resource.
- Object getContent()
  selects the appropriate content handler to read the

resource data and convert it into an object. This method is not useful for reading standard types such as text/plain or image/gif unless you install your own content handler.

4.3.3. Posting Form Data

In the preceding section, you saw how to read data from a web server. Now we will show you how your programs can send data back to a server.

To send information from a web browser to a server, a user fills out a form, like the one in Figure 4.6.

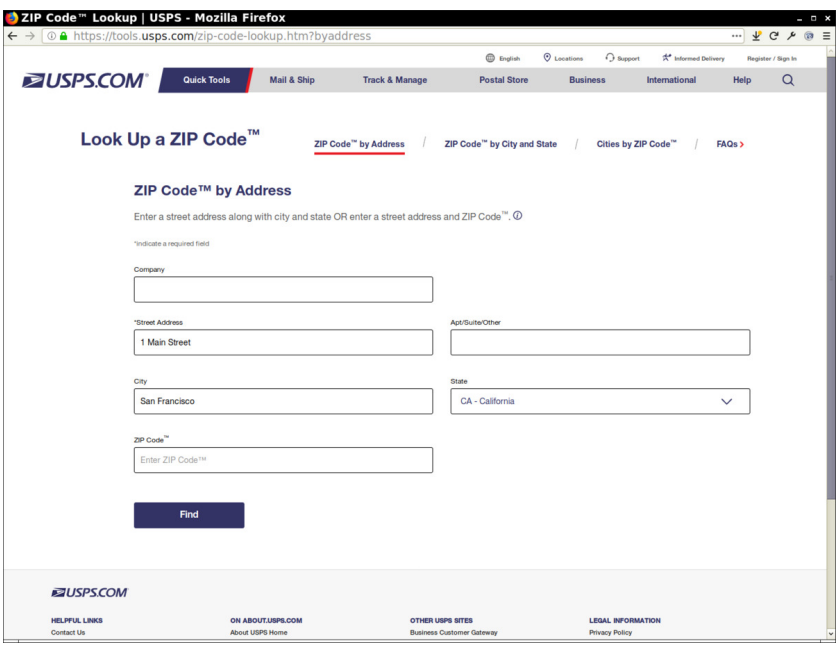


Figure 4.6: An HTML form

When the user clicks the Submit button, the text in the text fields and the settings of any checkboxes, radio buttons, and other input elements are sent back to the server. The server processes the submitted values.

The server then produces another HTML page and sends it back to the browser. This sequence is illustrated in Figure 4.7. The response page can contain new information (for example, in an information-search program) or just an acknowledgment. The web browser then displays the response page.

.

Figure 4.7: Data flow during execution of a form submission

We do not discuss the implementation of server-side programs in this book. Our interest is merely in writing client programs that interact with existing server-side programs.

Two commands, called GET and POST, are commonly used to send information to a server.

In the GET command, you simply attach query parameters to the end of the URL. The URL has the form

https://*host*/*path*?*query*

Each parameter has the form *name*=*value*. Parameters are separated by & characters. Parameter values are encoded using the URL encoding scheme, following these rules:

- Leave the characters A through Z, a through z, 0 through 9, and . - ~ _ unchanged.
- Replace all spaces with + characters.
- Encode all other characters into UTF-8 and encode each byte by a %, followed by a two-digit hexadecimal number.

For example, to transmit San Francisco, CA, you use San +Francisco%2c+CA, as the hexadecimal number 2c is the UTF-8 code of the ',' character.

This encoding keeps any intermediate programs from messing with spaces and other special characters.

At the time of this writing, the Google Maps site (https://

www.google.com/maps) accepts query parameters with names q and hl whose values are the location query and the human language of the response. To get a map of 1 Market Street in San Francisco, with a response in German, use the following URL:

```
https://www.google.com/maps?q=1+Market+Street+San+Francisco&h
```

Very long query strings can look unattractive in browsers, and older browsers and proxies have a limit on the number of characters that you can include in a GET request. For that reason, a POST request is often used for forms with a lot of data. In a POST request, you do not attach parameters to a URL; instead, you get an output stream from the URLConnection and write name/value pairs to the output stream. You still have to URL-encode the values and separate them with & characters.

Let us look at this process in detail. To post data to a server-side program, first establish a URLConnection:

```
URL url = new URI("https://host/path").toURL();
URLConnection connection = url.openConnection();
```

Then, call the setDoOutput method to set up the connection for output:

```
connection.setDoOutput(true);
```

Next, call getOutputStream to get a stream through which you can send data to the server. If you are sending text to the server, it is convenient to wrap that stream into a PrintWriter.

```
var out = new PrintWriter(connection.getOutputStream());
```

Now you are ready to send data to the server:

```
out.print(name1 + "=" + URLEncoder.encode(value1, StandardCharse
out.print(name2 + "=" + URLEncoder.encode(value2, StandardCharse
```

Close the output stream:

```
out.close();
```

Finally, call getInputStream and read the server response.

The static methods URLEncoder.encode and URLDecoder.decode have a version that uses the default charset, but they have been deprecated in Java 6 because they should always be used with UTF-8. Sadly, they are still deprecated even though Java 18 changed the default character encoding to UTF-8.

Let's run through a practical example. The web site at https://tools.usps.com/zip-code-lookup.htm?byaddress contains a form to find the zip code for a street address (see Figure 4.6). To use this form in a Java program, you need to know the URL and the parameters of the POST request.

You could get that information by looking at the HTML code of the form, but it is usually easier to "spy" on a request with a network monitor. Most browsers have a network monitor as part of their development toolkit. For example, Figure 4.8 shows a screen capture of the Firefox network monitor when submitting data to our example web site. You can find out the submission URL as well as the parameter names and values.

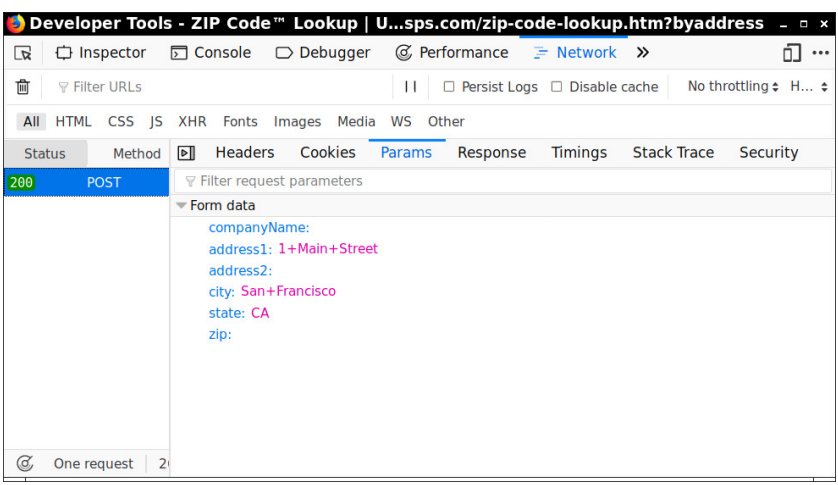


Figure 4.8: Monitoring the submission of a form

When posting form data, the HTTP header includes the content type:

```
Content-Type: application/x-www-form-urlencoded
```

You can also post data in other formats. For example, when sending data in JavaScript Object Notation (JSON), set the content type to application/json.

The header for a POST must also include the content length, for example

```
Content-Length: 124
```

The program in Listing 4.9 sends POST form data to any server-side program. Place the data into a .properties file such as the following:

```
url=https://tools.usps.com/tools/app/ziplookup/zipByAddress
User-Agent=HTTPie/0.9.2
address1=1 Market Street
address2=
city=San Francisco
state=CA
companyName=
. . .
```

The program removes the url and User-Agent entries and sends all others to the doPost method.

In the doPost method, we first open the connection and set the user agent. (The zip code service does not work with the default User-Agent request parameter which contains the string Java, perhaps because the postal service doesn't want to serve programmatic requests.)

Then we call setDoOutput(true), and open the output stream. We then enumerate all keys and values. For each of them, we send the key, = character, value, and & separator character:

```
out.print(key);
out.print("=");
```

```
out.print(URLEncoder.encode(value, StandardCharsets.UTF_8));
if (more pairs) out.print("&");
```

When switching from writing to reading any part of the response, the actual interaction with the server happens. The Content-Length header is set to the size of the output. The Content-Type header is set to application/x-www-form-urlencoded unless a different content type was specified. The headers and data are sent to the server. Then the response headers and server response are read and can be queried. In our example program, this switch happens in the call to connection.getContentEncoding().

There is one twist with reading the response. If a server-side error occurs, the call to connection.getInputStream() throws a FileNotFoundException. However, the server still sends an error page back to the browser (such as the ubiquitous “Error 404—page not found”). To capture this error page, call the getErrorStream method:

```
InputStream err = connection.getErrorStream();
```

> The getErrorStream method, as well as several other methods in this program, belong to the HttpURLConnection subclass of URLConnection. If you make a request to an URL that starts with http:// or https://, you can cast the resulting connection object to HttpURLConnection.

When you send POST data to a server, it can happen that the server-side program responds with a redirect: a different URL that should be called to get the actual information. The server could do that because the information is available elsewhere, or to provide a bookmarkable URL. The HttpURLConnection class can handle redirects in most cases.

> If cookies need to be sent from one site to another in a redirect, you can configure the global cookie handler like this:

CookieHandler.setDefault(new CookieManager(null, CookiePo

Then cookies will be properly included in the redirect.

Even though redirects are usually automatically handled, there are some situations where you need to do them yourself. Automatic redirects between HTTP and HTTPS are not supported for security reasons. Redirects can also fail for more subtle reasons. For example, an earlier version of the zip code service used a redirect. Recall that we set the User-Agent request parameter so that the post office didn't think we made a request via the Java API. While it is possible to set the user agent to a different string in the initial request, that setting is not used in automatic redirects. Instead, automatic redirects always send a generic user agent string that contains the word Java.

In such situations, you can manually carry out the redirects. Before connecting the server, turn off automatic redirects:

```
connection.setInstanceFollowRedirects(false);
```

After making the request, get the response code:

```
int responseCode = connection.getResponseCode();
```

Check if it is one of

```
HttpURLConnection.HTTP_MOVED_PERM
HttpURLConnection.HTTP_MOVED_TEMP
HttpURLConnection.HTTP_SEE_OTHER
```

In that case, get the Location response header to obtain the URL for the redirect. Then disconnect and make another connection to the new URL:

```
String location = connection.getHeaderField("Location");
if (location != null) {
   URL base = connection.getURL();
   connection.disconnect();
   URL url = base.toURI().resolve(location).toURL();
   connection = (HttpURLConnection) url.openConnection();
```

```
      . . .
   }
```

The techniques that this program illustrates can be useful whenever you need to query information from an existing web site. Simply find out the parameters that you need to send, and then strip out the HTML tags and other unnecessary information from the reply.

Listing 4.9 **v2ch04/post/PostDemo.java**

```
package v2ch04.post;

import module java.base;

/**
 * This program demonstrates how to use the URLConnection class for a POST
 */
public class PostDemo {
   void main(String[] args) throws Exception {
      String propsFilename = args.length > 0 ? args[0] : "v2ch04/post/post.proper
      var props = new Properties();
      try (Reader in = Files.newBufferedReader(Path.of(propsFilename))) {
         props.load(in);
      }
      String urlString = props.remove("url").toString();
      Object userAgent = props.remove("User-Agent");
      Object redirects = props.remove("redirects");
      CookieHandler.setDefault(new CookieManager(null, CookiePolicy.ACCEPT
      String result = doPost(new URI(urlString).toURL(), props,
            userAgent == null ? null : userAgent.toString(),
            redirects == null ? -1 : Integer.parseInt(redirects.toString()));
      IO.println(result);
   }

   /**
    * Do an HTTP POST.
    * @param url the URL to post to
    * @param nameValuePairs the query parameters
    * @param userAgent the user agent to use, or null for the default user agen
    * @param redirects the number of redirects to follow manually, or -1 for au
    *        redirects
```

```
 * @return the data returned from the server
 * @throws URISyntaxException
 */
String doPost(URL url, Map<Object, Object> nameValuePairs, String userAge
      throws IOException, URISyntaxException {
   var connection = (HttpURLConnection) url.openConnection();
   if (userAgent != null) connection.setRequestProperty("User-Agent", userAg

   if (redirects >= 0) connection.setInstanceFollowRedirects(false);

   connection.setDoOutput(true);

   try (var out = new PrintWriter(connection.getOutputStream())) {
      boolean first = true;
      for (Map.Entry<Object, Object> pair : nameValuePairs.entrySet()) {
         if (first)
            first = false;
         else
            out.print("&");
         String name = pair.getKey().toString();
         String value = pair.getValue().toString();
         out.print(name);
         out.print("=");
         out.print(URLEncoder.encode(value, StandardCharsets.UTF_8));
      }
   }
   String encoding = connection.getContentEncoding();
   if (encoding == null) encoding = "UTF-8";

   if (redirects > 0) {
      int responseCode = connection.getResponseCode();
      if (responseCode == HttpURLConnection.HTTP_MOVED_PERM
            || responseCode == HttpURLConnection.HTTP_MOVED_TEMP
            || responseCode == HttpURLConnection.HTTP_SEE_OTHER) {
         String location = connection.getHeaderField("Location");
         if (location != null) {
            URL base = connection.getURL();
            connection.disconnect();
            return doPost(base.toURI().resolve(location).toURL(), nameValuePai
                  userAgent, redirects - 1);
         }
      }
   }
   else if (redirects == 0) { throw new IOException("Too many redirects"); }
```

```
      var response = new StringBuilder();
      try (var in = new Scanner(connection.getInputStream(), encoding)) {
         while (in.hasNextLine()) {
            response.append(in.nextLine());
            response.append("\n");
         }
      }
      catch (IOException e) {
         InputStream err = connection.getErrorStream();
         if (err == null) throw e;
         try (var in = new Scanner(err)) {
            response.append(in.nextLine());
            response.append("\n");
         }
      }

      return response.toString();
   }
}
```

java.net.HttpURLConnection1.0

- InputStream getErrorStream()
  returns a stream from which you can read server error messages.

java.net.URLEncoder1.0

- static String encode(String s, String encoding)1.4
  returns the URL-encoded form of the string s, using the given character encoding scheme. (The recommended scheme is "UTF-8".) In URL encoding, the characters A–Z, a–z, 0–9, - _ . ~ are left unchanged. Space is encoded into +, and all other characters are encoded into sequences of encoded bytes of the form %*XY*, where 0x*XY* is the hexadecimal value of the byte.

java.net.URLDecoder1.2

- static string decode(String s, String encoding)1.4 returns the decoding of the URL encoded string s under the given character encoding scheme.

4.4. The HTTP Client

The URLConnection class was designed before HTTP was the universal protocol of the Web. It provides support for a number of protocols, but its HTTP support is somewhat cumbersome. When the decision was made to support HTTP/2, it became clear that it would be best to provide a modern client interface instead of reworking the existing API. The HttpClient provides HTTP/2 support and a more convenient API than the URLConnection class with its rather fussy set of stages.

4.4.1. The HttpClient Class

An HttpClient can issue requests and receive responses. You get a client by calling

```
HttpClient client = HttpClient.newHttpClient()
```

Alternatively, if you need to configure the client, you use a builder API, like this:

```
HttpClient client = HttpClient.newBuilder()
   .followRedirects(HttpClient.Redirect.ALWAYS)
   .build();
```

That is, you get a builder, call methods to customize the item that is going to be built, and then call the build method to finalize the building process. This is a common pattern for constructing immutable objects.

There are a number of advanced options for selecting HTTP 2 and for controlling cookies, SSL, proxy settings, and authentication.

The HttpClient is autocloseable, so you can declare it in a try-

with-resources statement. Its close method waits for the completion of submitted requests and then closes its connection pool.

4.4.2. The HttpRequest class and Body Publishers

You also follow the builder pattern for formulating requests. Here is a GET request:

```
HttpRequest request = HttpRequest.newBuilder()
   .uri(new URI(urlString))
   .GET()
   .build();
```

The URI is the “uniform resource identifier” which is, when using HTTP, the same as a URL. However, in Java, the URL class has methods for actually opening a connection to a URL, whereas the URI class is only concerned with the syntax (scheme, host, port, path, query, fragment, and so on).

With a POST request, you need a “body publisher” that turns the request data into the data that is being posted. There are body publishers for strings, byte arrays, and files. For example, if your request is in JSON, you just provide the JSON string to a string body publisher.

```
HttpRequest request = HttpRequest.newBuilder()
   .uri(new URI(urlString))
   .header("Content-Type", "application/json")
   .POST(HttpRequest.BodyPublishers.ofString(jsonString))
   .build();
```

It is unfortunate that the API does not support the required formatting for common content types. The sample program in Listing 4.10 provides body publishers for form data and file uploads.

In that program, you can see how body publishers are used to send strings and files. The file upload requires both, employing the BodyPublishers.concat method to concatenate multiple body publishers to produce the request body.

The HttpRequest.Builder class also has build methods for the less common PUT, DELETE, and HEAD requests.

Java 16 adds a builder for filtering the headers of an existing HttpRequest. You provide the request and a function that receives the header names and values, returning true for those that should be retained. For example, here we modify the content type:

```
HttpRequest request2 = HttpRequest.newBuilder(request, (name, val
      !name.equalsIgnoreCase("Content-Type")) // Remove old content
   .header("Content-Type", "application/xml") // Add new content typ
   .build();
```

4.4.3. The HttpResponse Interface and Body Handlers

When sending the request, you have to tell the client how to handle the response. If you just want the body as a string, send the request with a HttpResponse.BodyHandlers.ofString(), like this:

```
HttpResponse<String> response
   = client.send(request, HttpResponse.BodyHandlers.ofString());
```

HttpResponse is a generic interface whose type parameter denotes the type of the response body. You get the response body string simply as

```
String bodyString = response.body();
```

There are other response body handlers that get the response as a byte array or input stream. BodyHandlers.ofFile(filePath) yields a handler that saves the response to the given file, and BodyHandlers.ofFileDownload(directoryPath) saves the response in the given directory, using the file name from the Content-Disposition header. Finally, the handler obtained from BodyHandlers.discarding() simply discards the response.

Processing the contents of the response is not considered part of the API. For example, if you receive JSON data, you need a JSON library to parse the contents.

The HttpResponse object also yields the status code and the response headers.

```
int status = response.statusCode();
HttpHeaders responseHeaders = response.headers();
```

You can turn the HttpHeaders object into a map:

```
Map<String, List<String>> headerMap = responseHeaders.map();
```

The map values are lists since in HTTP, each key can have multiple values.

If you just want the value of a particular key, and you know that there won't be multiple values, call the firstValue method:

```
Optional<String> lastModified = responseHeaders.firstValue("Last-M
```

You get the response value or an empty optional if none was supplied.

> The HttpHeaders class is aware that keys for HTTP parameters are case-insensitive. If you call responseHeaders.firstValue("Last-Modified"), you get the desired header value even if it was sent as last-modified.

4.4.4. Asynchronous Processing

You can process the response asynchronously. When building the client, provide an executor:

```
ExecutorService executor = Executors.newCachedThreadPool();
HttpClient client = HttpClient.newBuilder().executor(executor).build
```

Build a request and then invoke the sendAsync method on the client. You receive a CompletableFuture<HttpResponse<T>>, where T is the type of the body handler. Use the CompletableFuture API as described in Chapter 10 of Volume I:

```
HttpRequest request = HttpRequest.newBuilder().uri(uri).GET().build
```

```
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
   thenAccept(response -> . . .);
```

To enable logging for the HttpClient, add this line to net.properties in your JDK:

```
jdk.httpclient.HttpClient.log=all
```

Instead of all, you can specify a comma-separated list of the following event types:

- channel
- content
- errors
- frames, optionally followed by one or more of :control, :data, :window, or :all
- headers
- requests
- ssl
- trace

Don't use any spaces:

```
jdk.httpclient.HttpClient.log=headers,requests,frames:control:
```

Then, set the logging level for the logger with name jdk.httpclient.HttpClient to INFO, for example by adding this line to the logging.properties file in your JDK:

```
jdk.httpclient.HttpClient.level=INFO
```

The program in Listing 4.10 shows how to make synchronous requests to different services. The class in Listing 4.11 implements the body publishers for forms, file upload, and JSON. Run the program as:

```
javac v2ch04/httpClient/HttpClientDemo.java
java v2ch04.httpClient.HttpClientDemo v2ch04/httpClient/form.prop
java v2ch04.httpClient.HttpClientDemo v2ch04/httpClient/fileupload
java v2ch04.httpClient.HttpClientDemo v2ch04/httpClient/json.prope
```

With each run, a “Hello, World” program is sent to the CodeCheck service, and the result of running it is displayed. That service can accept form posts, file uploads, and JSON posts. Note the logging output from the HttpClient.

Listing 4.10 **v2ch04/httpClient/HttpClientDemo.java**

```
package v2ch04.httpClient;

import module java.base;
import module java.net.http;
import java.net.http.HttpRequest.BodyPublisher;

/**
 * This program demonstrates the HTTP client
 */
class HttpClientDemo {
   void main(String[] args) throws Exception {
      System.setProperty("jdk.httpclient.HttpClient.log", "headers,requests,conte
      String propsFilename = args.length > 0 ? args[0] : "v2ch04/httpClient/json.p
      Path propsPath = Path.of(propsFilename);
      var props = new Properties();
      try (Reader in = Files.newBufferedReader(propsPath)) {
         props.load(in);
      }
      String urlString = "" + props.remove("url");
      String contentType = "" + props.remove("Content-Type");
      BodyPublisher publisher = null;
      if (contentType.equals("application/x-www-form-urlencoded"))
         publisher = MoreBodyPublishers.ofFormData(props);
      else if (contentType.equals("multipart/form-data")) {
         // Split each value along commas, replace strings starting with
         // file:// with Path objects
         var data = new HashMap<String, List<?>>();
         for (Map.Entry<Object, Object> entry : props.entrySet()) {
            data.put(entry.getKey().toString(),
               Stream.of(entry.getValue().toString().split("\\s*,\\s*"))
                  .map(s -> s.startsWith("file://")
                        ? propsPath.getParent().getParent()
                              .resolve(Path.of(s.substring(7)))
                        : s)
                  .toList());
```

```
         }
         String boundary = UUID.randomUUID().toString().replace("-", "");
         contentType += "; boundary=" + boundary;
         publisher = MoreBodyPublishers.ofMimeMultipartData(data, boundary
      }
      else {
         contentType = "application/json";
         publisher = MoreBodyPublishers.ofSimpleJSON(props);
      }

      String result = doPost(urlString, contentType, publisher);
      System.out.println(result);
   }

   String doPost(String url, String contentType, BodyPublisher publisher)
         throws IOException, URISyntaxException, InterruptedException {
      try (HttpClient client = HttpClient.newBuilder()
            .followRedirects(HttpClient.Redirect.ALWAYS).build()) {

         HttpRequest request = HttpRequest.newBuilder().uri(new URI(url))
               .header("Content-Type", contentType).POST(publisher).build();

         HttpResponse<String> response = client.send(request,
               HttpResponse.BodyHandlers.ofString());
         return response.body();
      }
   }
}
```

Listing 4.11 **v2ch04/httpClient/MoreBodyPublishers.java**

```
package v2ch04.httpClient;

import module java.base;
import java.net.http.HttpRequest.*;

public class MoreBodyPublishers {
   public static BodyPublisher ofFormData(Map<Object, Object> data) {
      boolean first = true;
      var builder = new StringBuilder();
      for (Map.Entry<Object, Object> entry : data.entrySet()) {
         if (first)
            first = false;
```

```
        else
            builder.append("&");
        builder.append(
                URLEncoder.encode(entry.getKey().toString(), StandardCharsets.UTF
        builder.append("=");
        builder.append(
                URLEncoder.encode(entry.getValue().toString(), StandardCharsets.U
    }
    return BodyPublishers.ofString(builder.toString());
}

public static BodyPublisher ofMimeMultipartData(Map<String, List<?>> data
        throws IOException {
    var bps = new ArrayList<BodyPublisher>();
    var header = new StringBuilder();
    for (Map.Entry<String, List<?>> entry : data.entrySet()) {
        for (Object value : entry.getValue()) {
            header.append("--%s\r\nContent-Disposition: form-data; name=%s"
                    .formatted(boundary, entry.getKey()));

            if (value instanceof Path path) {
                header.append("; filename=\"%s\"\r\n".formatted(path.getFileName(
                String mimeType = Files.probeContentType(path);
                if (mimeType != null)
                    header.append("Content-Type: %s\r\n".formatted(mimeType));
                header.append("\r\n");
                bps.add(BodyPublishers.ofString(header.toString()));
                bps.add(BodyPublishers.ofFile(path));
            }
            else {
                header.append("\r\n\r\n%s\r\n".formatted(value));
                header.append("\r\n");
                bps.add(BodyPublishers.ofString(header.toString()));
            }
            header = new StringBuilder("\r\n");
        }
    }
    bps.add(BodyPublishers.ofString("\r\n--%s--\r\n".formatted(boundary)));
    return BodyPublishers.concat(bps.toArray(BodyPublisher[]::new));
}

public static BodyPublisher ofSimpleJSON(Map<Object, Object> data) {
    var builder = new StringBuilder();
    builder.append("{");
```

```
        var first = true;
        for (Map.Entry<Object, Object> entry : data.entrySet()) {
            if (first)
                first = false;
            else
                builder.append(",");
            builder.append(jsonEscape(entry.getKey().toString())).append(": ")
                    .append(jsonEscape(entry.getValue().toString()));
        }
        builder.append("}");
        return BodyPublishers.ofString(builder.toString());
    }

    private static Map<Character, String> replacements = Map.of('\b', "\\b", '\f', "\\
            "\\n", '\r', "\\r", '\t', "\\t", '"', "\\"", '\\', "\\\\");

    private static StringBuilder jsonEscape(String str) {
        var result = new StringBuilder("\"");
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            String replacement = replacements.get(ch);
            if (replacement == null)
                result.append(ch);
            else
                result.append(replacement);
        }
        result.append("\"");
        return result;
    }
}
```

java.net.http.HttpClient11

- static HttpClient newHttpClient()
  yields an HttpClient with a default configuration.
- static HttpClient.Builder newBuilder()
  yields a builder for building an HttpClient.
- <T> HttpResponse<T> send(HttpRequest request,
  HttpResponse.BodyHandler<T> responseBodyHandler)
- <T> CompletableFuture<HttpResponse<T>>
  sendAsync(HttpRequest request,
  HttpResponse.BodyHandler<T> responseBodyHandler)
  make a synchronous or asynchronous request and process

the response body with the given handler.

java.net.http.HttpClient.Builder11

- HttpClient build()
  yields an HttpClient with the properties configured by this builder.
- HttpClient.Builder followRedirects(HttpClient.Redirect policy)
  sets the redirect policy to one of the values ALWAYS, NEVER, or NORMAL (only refuse redirects from HTTPS to HTTP) of the HttpClient.Redirect enumeration.
- HttpClient.Builder executor(Executor executor)
  sets the executor for asynchronous requests.

java.net.http.HttpRequest11

- HttpRequest.Builder newBuilder()
  returns a builder for building an HttpRequest.
- HttpRequest.Builder newBuilder(HttpRequest request, BiPredicate<String,String> filter)16
  returns a builder for building a request with the same properties as the provided one, with only the headers that are accepted by the filter.

java.net.http.HttpRequest.Builder11

- HttpRequest build()
  yields an HttpRequest with the properties configured by this builder.
- HttpRequest.Builder uri(URI uri)
  sets the URI for this request.
- HttpRequest.Builder header(String name, String value)
  sets a request header for this request.
- HttpRequest.Builder GET()
- HttpRequest.Builder DELETE()

- HttpRequest.Builder POST(HttpRequest.BodyPublisher bodyPublisher)
- HttpRequest.Builder PUT(HttpRequest.BodyPublisher bodyPublisher)
  set the request method and body for this request.

java.net.http.HttpResponse<T>[11]

- T body()
  yields the body of this response.
- int statusCode()
  yields the status code for this response.
- HttpHeaders headers()
  yields the response headers.

java.net.http.HttpHeaders[11]

- Map<String,List<String>> map()
  yields a Map of these headers.
- Optional<String> firstValue(String name)
  yields the first value with the given name in these headers, if present.

4.5. The Simple HTTP Server

Since Java 6, the JDK has provided a simple web server in the com.sun.net.httpserver package. In Java 18, a command-line tool was added to the JDK that invokes a static file server based upon that package. At that time, a few convenience classes were added, and JEP 408 affirms that the package is "officially supported." The web server is intended for testing, development, and debugging.

4.5.1. The Command-Line Tool

To start the simple file server, use the command

```
jwebserver
```

It serves all files in the current directory and its subdirectories on port 8000. When a file is served, a message is printed. You can change the directory and port with the -d and -p options:

```
jwebserver -d /tmp -p 8189
```

You can also set the message output level with the -o command-line option. Choices are info (the default), verbose, and none.

To kill the server, press Ctrl+C or use the kill/pkill command.

This server is only useful for serving files. It is similar to the python -m http.server command. In fact, you can also invoke the simple Java file server as java -m jdk.httpserver.

4.5.2. The HTTP Server API

You can start the simple file server programmatically as:

```
import com.sun.net.httpserver.*;
. . .
HttpServer server = SimpleFileServer.createFileServer(new InetSock
   Path.of("/tmp"), OutputLevel.VERBOSE);
server.start();
```

Now you can customize it in various ways. Alternatively, if you don't want to serve files, you can get a do-nothing server as

```
HttpServer server = HttpServer.create(new InetSocketAddress(port),
```

The second argument is the “backlog” of the server socket—that is, the maximum number of queued connection requests. With a zero argument, a default value is used.

You can set the executor for processing requests:

```
server.setExecutor(Executors.newVirtualThreadPerTaskExecutor());
```

For the server to do anything interesting, you need to create contexts.

A context has

- A path that must be a prefix of the request path
- A handler that computes the response
- A list of filters that can transform the request or response (see Section 4.5.4)

If a request matches multiple contexts, the one with the longest path is executed. For example, here we install two handlers for paths /static/ and /:

```
server.createContext("/static/", SimpleFileHandler.createFileHandler(
server.createContext("/", HttpHandlers.of(301,
   Headers.of("Location", "https://horstmann.com/corejava"), ""));
```

Note that you can leverage existing handlers, such as the handler of the simple file server.

The static HttpHandlers.of method produces a simple handler that returns a fixed response. You pass the status code, response headers, and response body.

The static HttpHandlers.handleOrElse uses a Predicate<Request> to select among two handlers:

```
HttpHandler handler = HttpHanders.handleOrElse(
   request -> request.getRequestMethod().equals("GET"),
   SimpleFileHandler.createFileHandler(staticRoot).
   HttpHandlers.of(500, Headers.of(), "Bad request method"))
```

For more complex handlers, you need to provide an instance of a subclass of HttpHandler. See the next section for details.

4.5.3. Handlers

A handler produces the response to an HTTP request in its method

```
public void handle(HttpExchange exchange)
```

The HttpExchange instance holds both the request and response data. The handler must use the object in this specific order:

1. Read the request method, URI, and headers:

   ```
   String method = exchange.getRequestMethod();
   URI uri = exchange.getRequestURI();
   Headers requestHeaders = exchange.getRequestHeaders();
   String userAgent = requestHeaders.getFirst("user-agent");
   ```

2. Read the request body for POST, PUT, and PATCH requests:

   ```
   InputStream in = exchange.getRequestBody();
   String body = new String(in.readAllBytes());
   in.close();
   ```

3. Set the response headers:

   ```
   Headers responseHeaders = exchange.getResponseHeaders();
   responseHeaders.add("Content-type", "text/plain");
   . . .
   ```

4. Set the status code and the length of the response body:

   ```
   exchange.sendResponseHeaders(200, responseBytes.length);
   ```

5. Write the response body, if there is one:

   ```
   OutputStream responseBody = exchange.getResponseBody();
   responseBody.write(responseBytes);
   responseBody.close();
   ```

The request and response headers are instances of the Headers class, which implements Map<String, List<String>>. Recall that there can be multiple values for each header.

The Headers class is similar to the HttpHeaders that you saw in Section 4.4, but it directly implements the Map<String,

List<String>> interface. As with HttpHeaders, the keys are case-insensitive.

If you know that there is at most one value for a given key, call getFirst to retrieve it. To add a key/value pair to a response map, use the add method, as in the code snippet above.

The sample program in Listing 4.12 has an example of a handler that produces a text response with all details of the request.

4.5.4. Filters

In addition to a handler, a context can have a chain of filters. A filter can modify the request or response, or carry out additional work such as logging or authentication. For example, the simple web server uses a filter for displaying the requests.

To implement a filter, extend the abstract Filter class and implement two methods. The description method should return some descriptive string. Typically, the doFilter method looks like this:

```
public void doFilter(HttpExchange exchange, Filter.Chain chain) {
   Request processing
   chain.doFilter(exchange);
   Response processing
}
```

By calling chain.doFilter(exchange), the remaining filters and the handler are invoked.

A filter can abort further filtering and handling—for example, if authentication fails. Then it should not call chain.doFilter(exchange), and instead produce a response in the exchange object.

You add a filter like this:

```
context.getFilters().add(filter);
```

> As you can see, the HTTP server API is a bit rough and old-fashioned. In your own code, do not provide methods that yield references to internal mutable data structures, as the Context.getFilters method does.

The Filter class has three static factory methods for filters:

```
public Filter beforeHandler(String description, Consumer<HttpExch
public Filter afterHandler(String description, Consumer<HttpExchar
public Filter adaptRequest(String description, UnaryOperator<Requ
```

The first two pass the exchange object (either before or after the request) to the consumer that can inspect or modify it.

The third method makes it a bit more convenient to add a header to a request:

```
Filter f = Filter.adaptRequest("Adding header",
   request -> request.with(headerName, List.of(headerValue)));
```

Of course, compared with mature web servers, such as Tomcat or Jetty, this server is rather basic. But it is a part of the JDK and can be sufficient for prototyping and testing.

The program in Listing 4.12 shows a simple server that echoes all request data in a plain-text response. Start it and point your browser to http://localhost:8189 to see the request headers of your browser. The server is also useful for observing the form post and the file upload requests of the program in Listing 4.10.

Listing 4.12 **v2ch04/httpServer/EchoServer.java**

```
package v2ch04.httpServer;

import module java.base;
import module jdk.httpserver;

/**
 * This program implements an HTTP server that listens to port 8189 or a port
 * argument, and echoes back all client input.
```

```
 */
class EchoServer {
   void main(String[] args) throws Exception {
      int port = args.length >= 1 ? Integer.parseInt(args[0]) : 8189;
      HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);
      server.setExecutor(Executors.newVirtualThreadPerTaskExecutor());
      server.createContext("/", HttpHandlers.of(301,
            Headers.of("Location", "https://horstmann.com/corejava"), ""));
      server.createContext("/echo", (HttpExchange exchange) -> {
         var headers = new StringBuilder();
         exchange.getRequestHeaders().forEach(
               (k, vs) -> vs.forEach(v -> headers.append("%s: %s\n".formatted(k, v)

         String requestBody = new String(exchange.getRequestBody().readAllByt
         String response = "%s %s\n%s\n%s\n".formatted(exchange.getRequestM
               exchange.getRequestURI(), headers, requestBody);
         byte[] responseBytes = response.getBytes();

         exchange.sendResponseHeaders(200, responseBytes.length);
         OutputStream responseBody = exchange.getResponseBody();
         responseBody.write(responseBytes);
         responseBody.close();
      });
      server.start();
   }
}
```

4.6. Sending E-Mail

In the past, it was simple to write a program that sends e-mail by making a socket connection to port 25, the SMTP port, of a mail server. The Simple Mail Transport Protocol (SMTP) describes the format for e-mail messages. Once you are connected to the server, send a mail header (in the SMTP format, which is easy to generate), followed by the mail message.

Here are the details:

1. Open a socket to your host.

   ```
   var s = new Socket("mail.yourserver.com", 25); // 25 is SMTP
   var out = new PrintWriter(s.getOutputStream());
   ```

2. Send the following information to the print stream:

   HELO *sending host*
   MAIL FROM: *sender e-mail address*
   RCPT TO: *recipient e-mail address*
   DATA
   Subject: *subject*
   (blank line)
   mail message (any number of lines)
   .
   QUIT

The SMTP specification (RFC 821) states that lines must be terminated with \r followed by \n.

It used to be that SMTP servers were often willing to route e-mail from anyone. However, in these days of spam floods, most servers have built-in checks and only accept requests from users or IP address ranges that they trust. Authentication usually happens over secure socket connections.

Implementing these authentication schemes manually would be very tedious. Instead, we will show you how to use the Jakarta Mail API to send e-mail from a Java program.

To use Jakarta Mail, you need to set up some properties that depend on your mail server. For example, with GMail, you use

```
mail.transport.protocol=smtps
mail.smtps.auth=true
mail.smtps.host=smtp.gmail.com
mail.smtps.user=accountname@gmail.com
```

Our sample program reads these from a property file.

For security reasons, we don't put the password into the property file but instead prompt for it.

> With most mail providers, you need to generate and use a special application password instead of using your

---

normal mail password.

Read in the property file, then get a mail session like this:

```
Session mailSession = Session.getDefaultInstance(props);
```

Make a message with the desired sender, recipient, subject, and message text:

```
var message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.TO, new InternetAddress(to));
message.setSubject(subject);
message.setText(builder.toString());
```

Then send it off:

```
Transport tr = mailSession.getTransport();
tr.connect(null, password);
tr.sendMessage(message, message.getAllRecipients());
tr.close();
```

The program reads the message from a text file of the format

Sender
Recipient
Subject
Message text (any number of lines)

At the time of this writing, GMail does not check the veracity of the information—you can supply any sender you like. (Keep this in mind the next time you get an e-mail message from president@whitehouse.gov inviting you to a black-tie affair on the front lawn.)

Because the program requires external JAR files, it is not contained in the corejava-bookcode.zip file, but instead in corejava-modulechapter.zip. You don't need to use modules to run the program. Simply compile and run as follows:

```
cd corejava-modulechapter/v2ch04.mail
javac -classpath .:../jars/\* com/horstmann/mail/MailDemo.java
```

```
java -classpath .:../jars/\* com.horstmann.mail.MailDemo path/to/mes
```

Or, if you prefer, use modules:

```
cd corejava-modulechapter/
javac -p v2ch04.mail:jars/jakarta.mail-api-2.1.3.jar:jars/jakarta.activa
   v2ch04.mail/com/horstmann/mail/MailDemo.java v2ch04.mail/mo
java -p v2ch04.mail:jars/jakarta.mail-api-2.1.3.jar:jars/jakarta.activat
:jars/angus-activation-2.0.2.jar:jars/angus-mail-2.0.3.jar \
   -m v2ch04.mail/com.horstmann.mail.MailDemo path/to/message.t
```

If you can't figure out why your mail connection isn't working, call

```
mailSession.setDebug(true);
```

and check out the messages. Also, the Jakarta Mail API FAQ at https://eclipse-ee4j.github.io/angus-mail/FAQ has some useful debugging hints.

In this chapter, you have seen how to write network clients and servers in Java and how to harvest information from web servers. The next chapter covers database connectivity. You will learn how to work with relational databases in Java, using the JDBC API.

Chapter 5 ■ Database Programming

In 1996, Sun released the first version of the JDBC API. This API lets programmers connect to a database to query or update it using the Structured Query Language (SQL). (SQL, usually pronounced “sequel,” is an industry standard for relational database access.) JDBC has since become one of the most commonly used APIs in the Java library.

JDBC has been updated several times. As this book is published, JDBC 4.3, the version included with Java 9, is the most current version.

In this chapter, I will explain the key ideas behind JDBC. I will introduce you to (or refresh your memory of) SQL, the industry-standard Structured Query Language for relational databases. The chapter has enough details and examples to let you start using JDBC for common programming situations.

> According to Oracle, JDBC is a trademarked term and not an acronym for Java Database Connectivity. It was named to be reminiscent of ODBC, a standard database API pioneered by Microsoft and since incorporated into the SQL standard.

5.1. The Design of JDBC

From the start, the developers of the Java technology were aware of the potential that Java showed for working with databases. In 1995, they began work on extending the standard Java library to deal with SQL access to databases. What they first hoped to do was to extend Java so that a program could talk to any random database using only “pure” Java. It didn’t take them long to realize that this is an impossible task: There are simply too many databases out there, using too many protocols. Moreover, although database vendors were all in favor of Java providing a standard network protocol for database access, they were only in favor of it if Java used their network protocol.

What all the database vendors and tool vendors did agree on was that it would be useful for Java to provide a pure Java API for SQL access along with a driver manager to allow third-party drivers to connect to specific databases. Database vendors could provide their own drivers to plug into the driver manager. There would then be a simple mechanism for registering third-party drivers with the driver manager.

This organization follows the very successful model of Microsoft’s ODBC which provided a C programming language interface for database access. Both JDBC and ODBC are based on the same idea: Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database.

This means the JDBC API is all that most programmers will ever have to deal with.

5.1.1. JDBC Driver Types

The JDBC specification classifies drivers into the following types:

- A type 1 driver translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Early

versions of Java included one such driver, the JDBC/ODBC bridge. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, many better drivers are available, and the JDK no longer provides the JDBC/ODBC bridge.

- A type 2 driver is written partly in Java and partly in native code; it communicates with the client API of a database. To use such a driver, you must install some platform-specific code onto the client in addition to a Java library.
- A type 3 driver is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This simplifies deployment because the platform-specific code is located only on the server.
- A type 4 driver is a pure Java library that translates JDBC requests directly to a database-specific protocol.

The JDBC specification is available at https://jcp.org/aboutJava/communityprocess/mrel/jsr221/index3.html.

Most database vendors supply either a type 3 or type 4 driver with their database. Furthermore, a number of third-party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers provided by the database vendors.

In summary, the ultimate goal of JDBC is to make possible the following:

- Programmers can write applications in the Java programming language to access any database using standard SQL statements (or even specialized extensions of SQL) while still following Java language conventions.
- Database vendors and database tool vendors can supply

the low-level drivers. Thus, they can optimize their drivers for their specific products.

If you are curious as to why Java just didn't adopt the ODBC model, the reason, as given at the JavaOne conference in 1996, was this:

- ODBC is hard to learn.
- ODBC has a few commands with lots of complex options. The preferred style in the Java programming language is to have simple and intuitive methods, but to have lots of them.
- ODBC relies on the use of void* pointers and other C features that are not natural in the Java programming language.
- An ODBC-based solution is inherently less safe and harder to deploy than a pure Java solution.

5.1.2. Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server (see Figure 5.1). In this model, a JDBC driver is deployed on the client.

.

Figure 5.1: A traditional client/server application

However, nowadays it is far more common to have a three-tier model where the client application does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates visual presentation (on the client) from the business logic (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java desktop application, a web browser, or a mobile app.

Communication between the client and the middle tier typically occurs through HTTP. JDBC manages the communication between the middle tier and the back-end database. Figure 5.2 shows the basic architecture.

.

Figure 5.2: A three-tier application

5.2. The Structured Query Language

JDBC lets you communicate with databases using SQL, which is the command language for essentially all relational database servers. Desktop databases such as Microsoft Access have a GUI for queries and updates, but JDBC is focused on server-based databases such as Oracle, PostgreSQL, MySQL, Microsoft SQL Server, and so on.

The JDBC package can be thought of as nothing more than an API for communicating SQL statements to databases. We will briefly introduce SQL in this section. If you have never seen SQL before, you might not find this material sufficient. If so, turn to one of the many learning resources on the topic; I recommend Learning SQL by Alan Beaulieu (O'Reilly, 2009) or the online book Learn SQL The Hard Way at https://sql.learncodethehardway.org.

You can think of a database as a bunch of named tables with rows and columns. Each column has a column name. Each row contains a set of related data.

As an example database for this book, we use a set of database tables that describe a collection of classic computer science books.

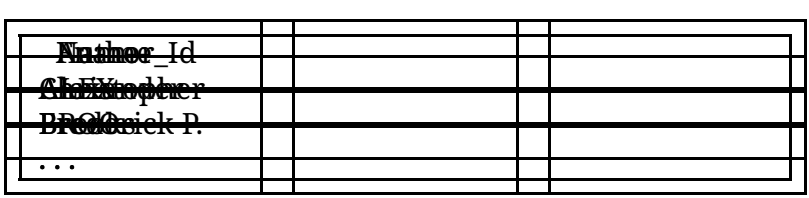


Table 5.1: The Authors Table

Author_Id	Name	Publisher_Id
1	Date, C. J.	1
2	Darwen, Hugh	1
3	Pascal, Fabian	2
...		

Table 5.2: The Books Table

ISBN	Title	Publisher_Id
0-201-96426-0	Database Design and Relational Theory: Normal Forms and All That the Rest of SQL	1
0-201-96426-0	Database Design and Relational Theory: Normal Forms and All That the Rest of SQL	1
0-11-551921-8	Database Systems: A Practical Approach to Design, Implementation and Management	3
...		

Table 5.3: The BooksAuthors Table

ISBN	Author_Id
0-201-96426-0	1
0-201-96426-0	2
0-11-551921-8	3
...	

Table 5.4: The Publishers Table

Publisher_Id	Name	URL
1	Addison Wesley	www.awl.com
2	John Wiley & Sons	www.wiley.com
...		

Figure 5.3 shows a view of the Books table. Figure 5.4 shows the result of joining this table with the Publishers table. The Books and the Publishers tables each contain an identifier for the publisher. When we join both tables on the publisher code, we obtain a query result made up of values from the joined tables. Each row in the result contains the information about a book, together with the publisher name and web page URL. Note that the publisher names and URLs are duplicated across several rows because we have several books with the same publisher.

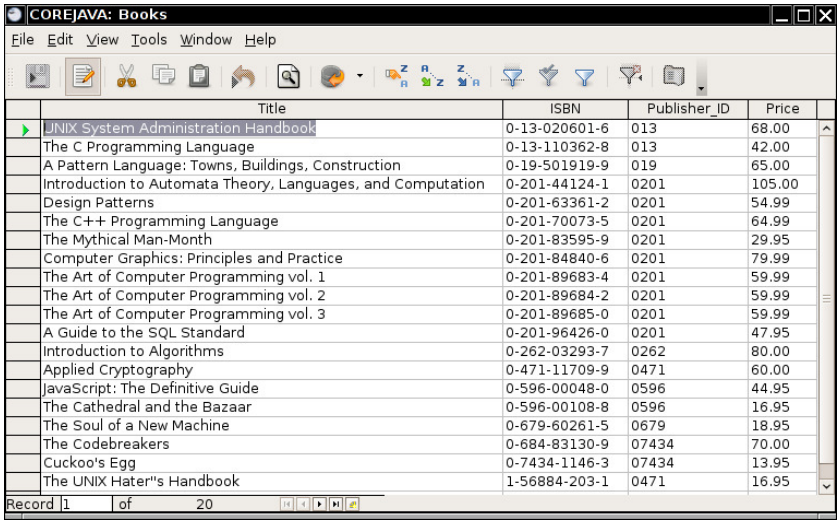

COREJAVA: Books
File Edit View Tools Window Help
Title ISBN Publisher_ID Price
UNIX System Administration Handbook 0-13-020601-6 013 68.00
The C Programming Language 0-13-110362-8 013 42.00
A Pattern Language: Towns, Buildings, Construction 0-19-501919-9 019 65.00
Introduction to Automata Theory, Languages, and Computation 0-201-44124-1 0201 105.00
Design Patterns 0-201-63361-2 0201 54.99
The C++ Programming Language 0-201-70073-5 0201 64.99
The Mythical Man-Month 0-201-83595-9 0201 29.95
Computer Graphics: Principles and Practice 0-201-84840-6 0201 79.99
The Art of Computer Programming vol. 1 0-201-89683-4 0201 59.99
The Art of Computer Programming vol. 2 0-201-89684-2 0201 59.99
The Art of Computer Programming vol. 3 0-201-89685-0 0201 59.99
A Guide to the SQL Standard 0-201-96426-0 0201 47.95
Introduction to Algorithms 0-262-03293-7 0262 80.00
Applied Cryptography 0-471-11709-9 0471 60.00
JavaScript: The Definitive Guide 0-596-00048-0 0596 44.95
The Cathedral and the Bazaar 0-596-00108-8 0596 16.95
The Soul of a New Machine 0-679-60261-5 0679 18.95
The Codebreakers 0-684-83130-9 07434 70.00
Cuckoo's Egg 0-7434-1146-3 07434 13.95
The UNIX Hater"s Handbook 1-56884-203-1 0471 16.95
Record 1 of 20

Figure 5.3: Sample table containing books

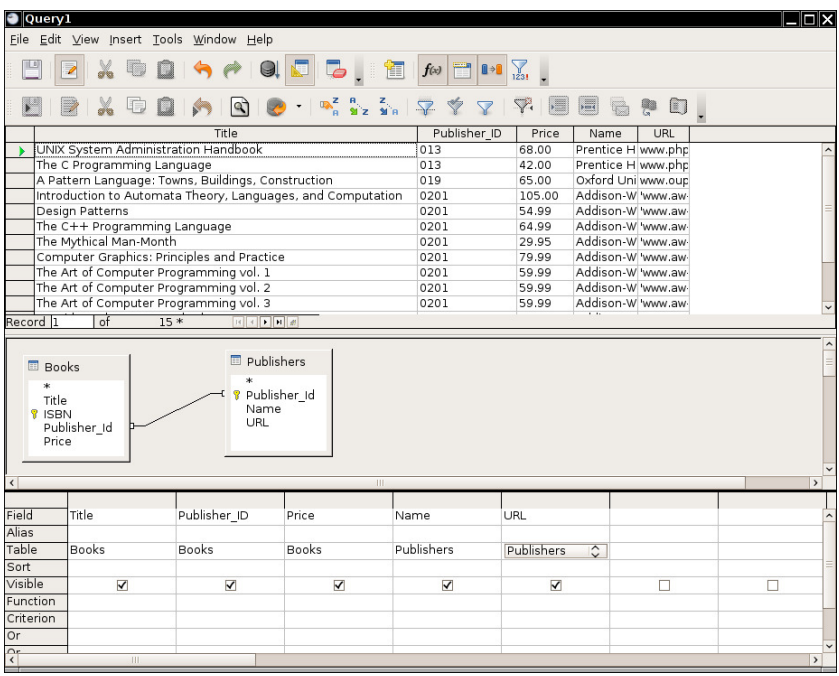

Query1
File Edit View Insert Tools Window Help
Title Publisher_ID Price Name URL
UNIX System Administration Handbook 013 68.00 Prentice H www.php
The C Programming Language 013 42.00 Prentice H www.php
A Pattern Language: Towns, Buildings, Construction 019 65.00 Oxford Uni www.oup
Introduction to Automata Theory, Languages, and Computation 0201 105.00 Addison-W www.aw-
Design Patterns 0201 54.99 Addison-W www.aw-
The C++ Programming Language 0201 64.99 Addison-W www.aw-
The Mythical Man-Month 0201 29.95 Addison-W www.aw-
Computer Graphics: Principles and Practice 0201 79.99 Addison-W www.aw-
The Art of Computer Programming vol. 1 0201 59.99 Addison-W www.aw-
The Art of Computer Programming vol. 2 0201 59.99 Addison-W www.aw-
The Art of Computer Programming vol. 3 0201 59.99 Addison-W www.aw-
Record 1 of 15 *
Books
*
Title
ISBN
Publisher_Id
Price
Publishers
*
Publisher_Id
Name
URL
Field Title Publisher_ID Price Name URL
Alias
Table Books Books Books Publishers Publishers
Sort
Visible
Function
Criterion
Or

Figure 5.4: Two tables joined together

The benefit of joining tables is avoiding unnecessary

duplication of data in the database tables. For example, a naive database design might have had columns for the publisher name and URL right in the Books table. But then the database itself, and not just the query result, would have many duplicates of these entries. If a publisher's web address changed, all entries would need to be updated. Clearly, this is somewhat error-prone. In the relational model, we distribute data into multiple tables so that no information is unnecessarily duplicated. For example, each publisher's URL is contained only once in the publisher table. If the information needs to be combined, the tables are joined.

In the figures, you can see a graphical tool to inspect and link the tables. Many vendors have tools to express queries in a simple form by connecting column names and filling information into forms. Such tools are often called query by example (QBE) tools. In contrast, a query that uses SQL is written out in text, using SQL syntax, for example:

```
SELECT Books.Title, Books.Publisher_Id, Books.Price, Publishers.Nam
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

In the remainder of this section, you will learn how to write such queries. If you are already familiar with SQL, just skip this section.

By convention, SQL keywords are written in capital letters, although this is not necessary.

The SELECT statement is quite flexible. You can simply select all rows in the Books table with the following query:

```
SELECT * FROM Books
```

The FROM clause is required in every SQL SELECT statement. It tells the database which tables to examine to find the data.

You can choose the columns that you want:

```
SELECT ISBN, Price, Title
```

```
FROM Books
```

You can restrict the rows in the answer with the WHERE clause:

```
SELECT ISBN, Price, Title
FROM Books
WHERE Price <= 29.95
```

Be careful with the “equals” comparison. SQL uses = and <>, rather than == or != as in the Java programming language, for equality testing.

> Some database vendors support the use of != for inequality testing. This is not standard SQL, so I recommend against such use.

The WHERE clause can also use pattern matching by means of the LIKE operator. The wildcard characters are not the usual * and ?, however. Use a % for zero or more characters and an underscore for a single character. For example,

```
SELECT ISBN, Price, Title
FROM Books
WHERE Title NOT LIKE '%n_x%'
```

excludes books with titles that contain words such as Unix or Linux.

Note that strings are enclosed in single quotes, not double quotes. A single quote inside a string is represented by a pair of single quotes. For example,

```
SELECT Title
FROM Books
WHERE Title LIKE '%''%'
```

reports all titles that contain a single quote.

You can select data from multiple tables:

```
SELECT * FROM Books, Publishers
```

Without a WHERE clause, this query is not very interesting. It lists all combinations of rows from both tables. In our case, where Books has 20 rows and Publishers has 8 rows, the result is a set of rows with 20 × 8 entries and lots of duplications. We really want to constrain the query to say that we are only interested in matching books with their publishers:

```
SELECT * FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

This query result has 20 rows, one for each book, because each book has one publisher in the Publisher table.

Whenever you have multiple tables in a query, the same column name can occur in two different places. That happened in our example. There is a column called Publisher_Id in both the Books and the Publishers tables. When an ambiguity would otherwise result, you must prefix each column name with the name of the table to which it belongs, such as Books.Publisher_Id.

You can use SQL to change the data inside a database as well. For example, suppose you want to reduce by $5.00 the current price of all books that have “C++” in their title:

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title LIKE '%C++%'
```

Similarly, to delete all C++ books, use a DELETE query:

```
DELETE FROM Books
WHERE Title LIKE '%C++%'
```

SQL comes with built-in functions for taking averages, finding maximums and minimums in a column, and so on, which we do not discuss here.

Typically, to insert values into a table, you can use the INSERT statement:

```
INSERT INTO Books
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

You need a separate INSERT statement for every row being inserted in the table.

Of course, before you can query, modify, and insert data, you must have a place to store data. Use the CREATE TABLE statement to make a new table. Specify the name and data type for each column. For example,

```
CREATE TABLE Books
(
  Title VARCHAR(60),
  ISBN CHAR(13),
  Publisher_Id VARCHAR(6),
  Price DECIMAL(10,2)
)
```

Table 5.5 shows the most common SQL data types.

Data Types
INTEGER, or INT
SMALLINT
NUMERIC(*m*,*n*), DECIMAL(*m*,*n*), or DEC(*m*,*n*)
FLOAT(*n*)
REAL
DOUBLE
CHARACTER(*n*) or CHAR(*n*)
VARCHAR(*n*)
BOOLEAN
DATE
TIME
TIMESTAMP
BLOB
CLOB

Table 5.5: Common SQL Data Types

In this book, we do not discuss the additional clauses, such as keys and constraints, that you can use with the CREATE TABLE statement.

5.3. JDBC Configuration

Of course, you need a database program for which a JDBC driver is available. There are many excellent choices, such as IBM DB2, Microsoft SQL Server, MySQL, Oracle, and PostgreSQL.

You must also create a database for your experimental use. We assume you name it COREJAVA. Create a new database, or have your database administrator create one with the appropriate permissions. You need to be able to create, update, and drop tables in the database.

If this is your first experience with databases, I recommend that you use the Apache Derby database, which is available from https://db.apache.org/derby.

Alternatively, if you are familiar with Docker, it is easy to run PostgreSQL in a docker image:

```
docker pull postgres
docker run -e POSTGRES_PASSWORD=secret -e POSTGRES_DB=CORE
```

You need to gather a number of items before you can write your first database program. The following sections cover these items.

5.3.1. Database URLs

When connecting to a database, you must use various database-specific parameters such as host names, port numbers, and database names.

JDBC uses a syntax similar to that of ordinary URLs to describe data sources. Here are examples of the syntax:

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

These JDBC URLs specify a Derby database and a PostgreSQL database named COREJAVA.

The general syntax is

```
jdbc:subprotocol:other stuff
```

where a subprotocol selects the specific driver for connecting to the database.

The format for the *other stuff* depends on the subprotocol used. You will need to look up your vendor's documentation for the specific format.

5.3.2. Driver JAR Files

You need to obtain the JAR file in which the driver for your database is located. If you use Derby, you need the file derbyclient.jar. With another database, you need to locate the appropriate driver. For example, the PostgreSQL drivers are available at https://jdbc.postgresql.org.

Include the driver JAR file on the class path when running a program that accesses the database. (You don't need the JAR file for compiling.)

When you launch programs from the command line, simply use the command

```
java -classpath driverJarPath:. ProgramName
```

On Windows, use a semicolon to separate the current directory (denoted by the . character) from the driver JAR path.

5.3.3. Starting the Database

The database server needs to be started before you can connect to it. The details depend on your database.

With the Derby database, follow these steps:

1. Open a command shell and change to a directory that will hold the database files.
2. Locate the file derbyrun.jar in the Derby installation directory. We will denote the directory containing lib/derbyrun.jar with *derby*.
3. Run the command

   ```
   java -jar derby/lib/derbyrun.jar server start
   ```

4. Double-check that the database is working correctly. Create a file ij.properties that contains these lines:

   ```
   ij.driver=org.apache.derby.jdbc.ClientDriver
   ij.protocol=jdbc:derby://localhost:1527/
   ij.database=COREJAVA;create=true
   ```

   From another command shell, run Derby's interactive scripting tool (called ij) by executing

   ```
   java -jar derby/lib/derbyrun.jar ij -p ij.properties
   ```

   Now you can issue SQL commands such as

   ```
   CREATE TABLE Greetings (Message VARCHAR(20));
   INSERT INTO Greetings VALUES ('Hello, World!');
   SELECT * FROM Greetings;
   DROP TABLE Greetings;
   ```

   Note that each command must be terminated by a semicolon. To exit, type

   ```
   EXIT;
   ```

5. When you are done using the database, stop the server with the command

   ```
   java -jar derby/lib/derbyrun.jar server shutdown
   ```

If you use another database, you need to consult the documentation to find out how to start and stop your database

server, and how to connect to it and issue SQL commands.

5.3.4. Connecting to the Database

In your Java program, open a database connection like this:

```
String url = "jdbc:postgresql:COREJAVA";
String username = "dbuser";
String password = "secret";
Connection conn = DriverManager.getConnection(url, username, pas
```

The driver manager iterates through the registered drivers to find a driver that can use the subprotocol specified in the database URL.

The getConnection method returns a Connection object. In the following sections, you will see how to use the Connection object to execute SQL statements.

To connect to the database, you will need a username and password for your database.

> By default, Derby lets you connect with any username, and it does not check passwords. A separate set of tables is generated for each user. The default username is app.

The test program in Listing 5.1 puts these steps to work. It loads connection parameters from a file named database.properties and connects to the database. The database.properties file supplied with the sample code contains connection information for the Derby database. If you use a different database, put your database-specific connection information into that file. Here is an example for connecting to a PostgreSQL database:

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

After connecting to the database, the test program executes the

following SQL statements:

```
CREATE TABLE Greetings (Message VARCHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

The result of the SELECT statement is printed, and you should see an output of

```
Hello, World!
```

Then the table is removed by executing the statement

```
DROP TABLE Greetings
```

To run this test, start your database as described previously, and launch the program as

```
java -classpath driverJarPath:. test.TestDB
```

One way to debug JDBC-related problems is to enable JDBC tracing. Call the DriverManager.setLogWriter method to send trace messages to a PrintWriter. The trace output contains a detailed listing of the JDBC activity. Most JDBC driver implementations provide additional mechanisms for tracing. For example, with Derby, you can add a traceFile option to the JDBC URL: jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out.

Since JDBC 4, driver JAR files contain a file META-INF/services/java.sql.Driver which contains the name of the driver class.

In the unlikely case that you use an older driver JAR, you need to find out the name of the JDBC driver class used by your vendor. Then set the jdbc.drivers property on the command line:

```
java -Djdbc.drivers=driverClassName ProgramName
```

Listing 5.1 **v2ch05/test/TestDB.java**

```
package v2ch05.test;

import module java.base;
import module java.sql;
import v2ch05.exec.ExecSQL;

/**
 * This program tests that the database and the JDBC driver are correctly confi
 */
class TestDB {
   void main() throws Exception {
      try {
         runTest();
      }
      catch (SQLException e) {
         for (Throwable t : e)
            t.printStackTrace();
      }
   }

   /**
    * Runs a test by creating a table, adding a value, showing the table contents
    * removing the table.
    */
   void runTest() throws SQLException, IOException {
      try (Connection conn = ExecSQL.getConnection()) {
         Statement stat = conn.createStatement();
         stat.executeUpdate("CREATE TABLE Greetings (Message VARCHAR(20))"
         stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World!')");

         try (ResultSet result = stat.executeQuery("SELECT * FROM Greetings")) {
            if (result.next()) System.out.println(result.getString(1));
         }
         stat.executeUpdate("DROP TABLE Greetings");
      }
   }
}
```

java.sql.DriverManager1.1

- static Connection getConnection(String url, String user, String password)
  establishes a connection to the given database and returns a Connection object.

5.4. Working with JDBC Statements

In the following sections, you will see how to use the JDBC Statement to execute SQL statements, obtain results, and deal with errors. Then we show you a simple program for populating a database.

5.4.1. Executing SQL Statements

To execute a SQL statement, you first create a Statement object. To create statement objects, use the Connection object that you obtained from the call to DriverManager.getConnection.

```
Statement stat = conn.createStatement();
```

Next, place the statement that you want to execute into a string, for example:

```
String command = """
UPDATE Books
SET Price = Price - 5.00
WHERE Title NOT LIKE '%Introduction%'
""";
```

Then, call the executeUpdate method of the Statement interface:

```
stat.executeUpdate(command);
```

The executeUpdate method returns a count of the rows that were affected by the SQL statement, or zero for statements that do not return a row count. For example, the call to executeUpdate in the preceding example returns the number of rows where the price was lowered by $5.00.

The executeUpdate method can execute actions such as INSERT, UPDATE, and DELETE, as well as data definition statements such as CREATE TABLE and DROP TABLE. However, you need to use the executeQuery method to execute SELECT queries. There is also a catch-all execute statement to execute arbitrary SQL statements. It's commonly used only for queries that a user supplies interactively.

When you execute a query, you are interested in the result. The executeQuery method returns an object of type ResultSet that you can use to walk through the result one row at a time.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

The basic loop for analyzing a result set looks like this:

```
while (rs.next()) {
   look at a row of the result set
}
```

The iteration protocol of the ResultSet interface is subtly different from that of the java.util.Iterator interface. Here, the iterator is initialized to a position before the first row. You must call the next method once to move the iterator to the first row. Also, there is no hasNext method; keep calling next until it returns false.

The order of the rows in a result set is completely arbitrary. Unless you specifically ordered the result with an ORDER BY clause, you should not attach any significance to the row order.

When inspecting an individual row, you will want to know the contents of the fields. A large number of accessor methods give you this information.

```
String isbn = rs.getString(1);
double price = rs.getDouble("Price");
```

There are accessors for various types, such as getString and getDouble. Each accessor has two forms, one taking a numeric

argument, and the other, a string argument. When you supply a numeric argument, you refer to the column with that number. For example, rs.getString(1) returns the value of the first column in the current row.

Unlike array indexes, database column numbers start at 1.

When you supply a string argument, you refer to the column in the result set with that name. For example, rs.getDouble("Price") returns the value of the column with label Price. Using the numeric argument is a bit more efficient, but string arguments make the code easier to read and maintain.

Each get method makes reasonable type conversions when the type of the method doesn't match the type of the column. For example, the call rs.getString("Price") converts the floating-point value of the Price column to a string.

java.sql.Connection **1.1**

- Statement createStatement()
  creates a Statement object that can be used to execute SQL queries and updates without parameters.
- void close()
  immediately closes the current connection and the JDBC resources that it created.

java.sql.Statement **1.1**

- ResultSet executeQuery(String sqlQuery)
  executes the SQL statement given in the string and returns a ResultSet object to view the query result.
- int executeUpdate(String sqlStatement)
- long executeLargeUpdate(String sqlStatement) **8**
  execute the SQL INSERT, UPDATE, or DELETE statement specified by the string. Also execute Data Definition

Language (DDL) statements such as CREATE TABLE. Return the number of rows affected, or 0 for a statement without an update count.

- boolean execute(String sqlStatement)
  executes the SQL statement specified by the string. Multiple result sets and update counts may be produced. Returns true if the first result is a result set, false if the result is an update count. Call getResultSet or getUpdateCount to retrieve the first result. See for details on processing multiple results.
- ResultSet getResultSet()
  returns the result set of the preceding query statement, or null if the preceding statement did not have a result set. Call this method only once per executed statement.
- int getUpdateCount()
- long getLargeUpdateCount()8
  return the number of rows affected by the preceding update statement, or -1 if the preceding statement was a statement without an update count. Call this method only once per executed statement.
- void close()
  closes this statement object and its associated result set.
- boolean isClosed()6
  returns true if this statement is closed.
- void closeOnCompletion()7
  causes this statement to be closed once all of its result sets have been closed.

java.sql.ResultSet1.1

- boolean next()
  makes the current row in the result set move forward by one. Returns false after the last row. Note that you must call this method to advance to the first row.
- *Xxx* get*Xxx*(int columnNumber)
- *Xxx* get*Xxx*(String columnLabel)
  (*Xxx* is a type such as int, double, String, Date, etc.)
- <T> T getObject(int columnIndex, Class<T> type)7
- <T> T getObject(String columnLabel, Class<T> type)7

- void updateObject(int columnIndex, Object x, SQLType targetSqlType)[8]
- void updateObject(String columnLabel, Object x, SQLType targetSqlType)[8]
  return or update the value of the column with the given column index or label, converted to the specified type. The column label is the label specified in the SQL AS clause or the column name if AS is not used.
- int findColumn(String columnName)
  gives the column index associated with a column name.
- void close()
  immediately closes the current result set.
- boolean isClosed()[6]
  returns true if this statement is closed.

5.4.2. Managing Connections, Statements, and Result Sets

Every Connection object can create one or more Statement objects. You can use the same Statement object for multiple unrelated commands and queries. However, a statement has at most one open result set. If you issue multiple queries whose results you analyze concurrently, you need multiple Statement objects.

Be forewarned, though, that there is a limit to the number of statements per connection. Use the getMaxStatements method of the DatabaseMetaData interface to find out the number of concurrently open statements that your JDBC driver supports.

In practice, you should probably not fuss with multiple concurrent result sets. If the result sets are related, you should be able to issue a combined query and analyze a single result. It is much more efficient to let the database combine queries than it is for a Java program to iterate through multiple result sets.

Be sure to finish processing of any result set before you issue a new query or update on a Statement object. The result sets of prior queries are automatically closed.

When you are done using a ResultSet, Statement, or Connection, it is a good idea to call the close method immediately. These objects use large data structures that draw on the finite resources of the database server.

The close method of a Statement object closes any associated result sets. Similarly, the close method of the Connection class closes all statements of the connection.

Conversely, you can call the closeOnCompletion method on a Statement, and it will close automatically as soon as all its result sets have closed.

If your connections are short-lived, don't worry about closing statements and result sets. To make sure a connection object cannot possibly remain open, use a try-with-resources statement:

```
try (Connection conn = . . .) {
   Statement stat = conn.createStatement();
   ResultSet result = stat.executeQuery(queryString);
   process query result
}
```

5.4.3. Analyzing SQL Exceptions

Each SQLException has a chain of SQLException objects that are retrieved with the getNextException method. This exception chain is in addition to the "cause" chain of Throwable objects that every exception has. (See Chapter 7 of Volume I for details about Java exceptions.) One would need two nested loops to fully enumerate all these exceptions. Fortunately, the SQLException class has been enhanced to implement the Iterable<Throwable> interface. The iterator() method yields an Iterator<Throwable> that iterates through both chains: It starts by going through the cause chain of the first SQLException, then moves on to the next SQLException, and so on. You can simply use an enhanced for loop:

```
for (Throwable t : sqlException) {
   do something with t
```

```
}
```

You can call getSQLState and getErrorCode on a SQLException to analyze it further. The first method yields a string that is standardized by either X/Open or SQL:2003. (Call the getSQLStateType method of the DatabaseMetaData interface to find out which standard is used by your driver.) The error code is vendor-specific.

The SQL exceptions are organized into an inheritance tree (shown in Figure 5.5). This allows you to catch specific error types in a vendor-independent way.

.

Figure 5.5: SQL exception types

In addition, the database driver can report nonfatal conditions as warnings. You can retrieve warnings from connections, statements, and result sets. The SQLWarning class is a subclass of SQLException (even though a SQLWarning is not thrown as an exception). Call getSQLState and getErrorCode to get further information about the warnings. Similar to SQL exceptions, warnings are chained. To retrieve all warnings, use this loop:

```
SQLWarning w = stat.getWarning();
while (w != null) {
   do something with w
   w = w.nextWarning();
}
```

The DataTruncation subclass of SQLWarning is used when data being read from the database are unexpectedly truncated. If data truncation happens in an update statement, a DataTruncation is thrown as an exception.

java.sql.SQLException1.1
■ SQLException getNextException()

gets the next SQL exception chained to this one, or null at the end of the chain.

- Iterator<Throwable> iterator()6
  gets an iterator that yields the chained SQL exceptions and their causes.
- String getSQLState()
  gets the "SQL state"—a standardized error code.
- int getErrorCode()
  gets the vendor-specific error code.

java.sql.SQLWarning1.1

- SQLWarning getNextWarning()
  returns the next warning chained to this one, or null at the end of the chain.

java.sql.Connection1.1java.sql.Statement1.1java.sql.ResultSet1.1

- SQLWarning getWarnings()
  returns the first of the pending warnings, or null if no warnings are pending.

java.sql.DataTruncation1.1

- boolean getParameter()
  returns true if the data truncation applies to a parameter, false if it applies to a column.
- int getIndex()
  returns the index of the truncated parameter or column.
- int getDataSize()
  returns the number of bytes that should have been transferred, or -1 if the value is unknown.
- int getTransferSize()
  returns the number of bytes that were actually transferred, or -1 if the value is unknown.

5.4.4. Populating a Database

We are now ready to write our first real JDBC program. Sure, it would be nice to try some of the fancy queries discussed earlier, but we have a problem: Right now, there are no data in the database. We need to populate the database, and a simple way of doing that is with a set of SQL instructions to create tables and insert data into them. Most database programs can process a set of SQL instructions from a text file, but there are pesky differences about statement terminators and other syntactical issues.

For that reason, we will use JDBC in a simple program that reads a file with SQL instructions, one instruction per line, and executes them.

Specifically, the program reads data from a text file in a format such as

```
CREATE TABLE Publishers (Publisher_Id VARCHAR(6), Name VARCHA
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www.aw
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', 'www.w
. . .
```

Listing 5.2 contains the code for the program that reads a file with SQL statements and executes them. You don't have to read through the code; simply use the program so you can populate your database and run the examples in the remainder of this chapter.

Make sure that your database server is running, and run the program as follows:

```
java -classpath driverJarPath:. v2ch05.exec.ExecSQL v2ch05/Books.sql
java -classpath driverJarPath:. v2ch05.exec.ExecSQL v2ch05/Authors.s
java -classpath driverJarPath:. v2ch05.exec.ExecSQL v2ch05/Publisher
java -classpath driverJarPath:. v2ch05.exec.ExecSQL v2ch05/BooksAut
```

Before running the program, check that the file database.properties is set up properly for your environment

(see Section 5.3.4).

Your database may also have a utility to process SQL files. For example, with Derby, you can run

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties v2ch05/Books
```

(The ij.properties file is described in Section 5.3.3.)

In the data format for the ExecSQL command, we allow an optional semicolon at the end of each line because most database utilities expect this format.

The following steps briefly describe the ExecSQL program:

1. Connect to the database. The getConnection method reads the properties in the file database.properties and adds the jdbc.drivers property to the system properties. The driver manager uses the jdbc.drivers property to load the appropriate database driver. The getConnection method uses the jdbc.url, jdbc.username, and jdbc.password properties to open the database connection.
2. Open the file with the SQL statements. If no file name was supplied, prompt the user to enter the statements on the console.
3. Execute each statement with the generic execute method. If it returns true, the statement had a result set. The four SQL files provided for the book database all end in a SELECT * statement so that you can see that the data were successfully inserted.
4. If there was a result set, print out the result. Since this is a generic result set, we need to use metadata to find out how many columns the result has. For more information, see Section 5.8.
5. If there is any SQL exception, print the exception and any chained exceptions that may be contained in it.
6. Close the connection to the database.

Listing 5.2 shows the code for the program.

Listing 5.2 **v2ch05/exec/ExecSQL.java**

```
package v2ch05.exec;

import module java.base;
import module java.sql;

/**
 * Executes all SQL statements in a file or the console.
 */
public class ExecSQL {
   void main(String[] args) throws Exception {
      try (Scanner in = args.length == 0 ? new Scanner(System.in)
            : new Scanner(Path.of(args[0]));
         Connection conn = getConnection()) {

         Statement stat = conn.createStatement();
         while (true) {
            if (args.length == 0) IO.println("Enter command or EXIT to exit:");

            if (!in.hasNextLine()) return;

            String line = in.nextLine().strip();
            if (line.equalsIgnoreCase("EXIT")) return;
            if (line.endsWith(";")) // remove trailing semicolon
               line = line.substring(0, line.length() - 1);
            try {
               boolean isResult = stat.execute(line);
               if (isResult) {
                  try (ResultSet rs = stat.getResultSet()) {
                     showResultSet(rs);
                  }
               } else {
                  int updateCount = stat.getUpdateCount();
                  IO.println(updateCount + " rows updated");
               }
            }
            catch (SQLException e) {
               for (Throwable t : e)
                  t.printStackTrace();
            }
         }
      }
      catch (SQLException e) {
```

```
      for (Throwable t : e)
         t.printStackTrace();
   }
}

/**
 * Gets a connection from the properties specified in the file database.proper
 * @return the database connection
 */
public static Connection getConnection() throws SQLException, IOException
   var props = new Properties();
   try (Reader in = Files.newBufferedReader(
         Path.of("v2ch05", "database.properties"))) {
      props.load(in);
   }
   String drivers = props.getProperty("jdbc.drivers");
   if (drivers != null) System.setProperty("jdbc.drivers", drivers);

   String url = props.getProperty("jdbc.url");
   String username = props.getProperty("jdbc.username");
   String password = props.getProperty("jdbc.password");

   return DriverManager.getConnection(url, username, password);
}

/**
 * Prints a result set.
 * @param result the result set to be printed
 */
void showResultSet(ResultSet result) throws SQLException {
   ResultSetMetaData metaData = result.getMetaData();
   int columnCount = metaData.getColumnCount();

   for (int i = 1; i <= columnCount; i++) {
      if (i > 1) IO.print(", ");
      IO.print(metaData.getColumnLabel(i));
   }
   IO.println();

   while (result.next()) {
      for (int i = 1; i <= columnCount; i++) {
         if (i > 1) IO.print(", ");
         IO.print(result.getString(i));
      }
```

```
            IO.println();
        }
    }
}
```

5.5. Query Execution

In this section, we write a program that executes queries against the COREJAVA database. For this program to work, you must have a COREJAVA database populated with tables as described in the preceding section.

When querying the database, you can select the author and the publisher or leave either of them as Any.

You can also change the data in the database. Select a publisher and type an amount. All prices of that publisher are adjusted by the amount you entered, and the program displays how many rows were changed. After a price change, you might want to run a query to verify the new prices.

5.5.1. Prepared Statements

In this program, we use one new feature, prepared statements. Consider the SQL query for all books by a particular publisher, regardless of the author:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = the name from the list box
```

Instead of building a separate query statement every time the user launches such a query, we can prepare a query with a host variable and use it many times, each time filling in a different string for the variable. That technique improves performance. Whenever the database executes a query, it first computes a strategy of how to do it efficiently. By preparing the query and reusing it, you ensure that the planning step is done only once.

Each host variable in a prepared query is indicated with a ?. If

there is more than one variable, you must keep track of the positions of the ? when setting the values. For example, our prepared query becomes

```
String publisherQuery = """
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishe
""";
PreparedStatement pstat = conn.prepareStatement(publisherQuery)
```

Before executing the prepared statement, you must bind the host variables to actual values with a set method. As with the get methods of the ResultSet interface, there are different set methods for the various types. Here, we want to set a string to a publisher name:

```
pstat.setString(1, publisher);
```

The first argument is the position number of the host variable that we want to set. The position 1 denotes the first ?. The second argument is the value that we want to assign to the host variable.

If you reuse a prepared query that you have already executed, all host variables stay bound unless you change them with a set method or call the clearParameters method. That means you only need to call a set*Xxx* method on those host variables that change from one query to the next.

Once all variables have been bound to values, you can execute the prepared statement:

```
ResultSet rs = pstat.executeQuery();
```

> Building a query manually, by concatenating strings, is tedious and potentially dangerous. You have to worry about special characters such as quotes—and, if your query involves user input, you have to guard against injection attacks. Therefore, use prepared statements

---

whenever your query involves variables.

The price update feature is implemented as an UPDATE statement. Note that we call executeUpdate, not executeQuery, because the UPDATE statement does not return a result set. The return value of executeUpdate is the count of changed rows.

```
int r = stat.executeUpdate();
IO.println(r + " rows updated");
```

A PreparedStatement object becomes invalid after the associated Connection object is closed. However, many databases automatically cache prepared statements. If the same query is prepared twice, the database simply reuses the query strategy. Therefore, don't worry about the overhead of calling prepareStatement.

The following list briefly describes the structure of the example program:

- The author and publisher array lists are populated by running two queries that return all author and publisher names in the database.
- The queries involving authors are complex. A book can have multiple authors, so the BooksAuthors table stores the correspondence between authors and books. For example, the book with ISBN 0-201-96426-0 has two authors with codes DATE and DARW. The BooksAuthors table has the rows

  0-201-96426-0, DATE, 1
  0-201-96426-0, DARW, 2

  to indicate this fact. The third column lists the order of the authors. (We can't just use the position of the rows in the table. There is no fixed row ordering in a relational table.) Thus, the query has to join the Books, BooksAuthors, and Authors tables to compare the author name with the one selected by the user.

```
SELECT Books.Price, Books.Title FROM Books, BooksAuthors, A
WHERE Authors.Author_Id = BooksAuthors.Author_Id AND Bo
AND Books.Publisher_Id = Publishers.Publisher_Id AND Autho
AND Publishers.Name = ?
```

Some Java programmers avoid complex SQL statements such as this one. A surprisingly common, but very inefficient, workaround is to write lots of Java code that iterates through multiple result sets. But the database is a lot better at executing query code than a Java program can be—that's the core competency of a database. A rule of thumb: If you can do it in SQL, don't do it in Java.

- The changePrices method executes an UPDATE statement. Note that the WHERE clause of the UPDATE statement needs the publisher code and we only know the publisher name. This problem is solved with a nested subquery:

```
UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Pub
```

Listing 5.3 is the complete program code.

Listing 5.3 **v2ch05/query/QueryDemo.java**

```
package v2ch05.query;

import module java.base;
import module java.sql;
import v2ch05.exec.ExecSQL;

/**
 * This program demonstrates several complex database queries.
 */
class QueryDemo {
   String allQuery = "SELECT Books.Price, Books.Title FROM Books";

   String authorPublisherQuery = """
```

```
SELECT Books.Price, Books.Title
FROM Books, BooksAuthors, Authors, Publishers
WHERE Authors.Author_Id = BooksAuthors.Author_Id
   AND BooksAuthors.ISBN = Books.ISBN
   AND Books.Publisher_Id = Publishers.Publisher_Id
   AND Authors.Name = ?
   AND Publishers.Name = ?
""";

   String authorQuery = """
SELECT Books.Price, Books.Title FROM Books, BooksAuthors, Authors
WHERE Authors.Author_Id = BooksAuthors.Author_Id"
   AND BooksAuthors.ISBN = Books.ISBN"
   AND Authors.Name = ?
""";

   String publisherQuery = """
SELECT Books.Price, Books.Title FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
   AND Publishers.Name = ?
""";

   String priceUpdate = """
UPDATE Books SET Price = Price + ? "
WHERE Books.Publisher_Id =
   (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
""";

   ArrayList<String> authors = new ArrayList<>();
   ArrayList<String> publishers = new ArrayList<>();

   void main() throws Exception {
      try (Connection conn = ExecSQL.getConnection()) {
         authors.add("Any");
         publishers.add("Any");
         try (Statement stat = conn.createStatement()) {
            // Fill the authors array list
            String query = "SELECT Name FROM Authors";
            try (ResultSet rs = stat.executeQuery(query)) {
               while (rs.next())
                  authors.add(rs.getString(1));
            }

            // Fill the publishers array list
```

```
            query = "SELECT Name FROM Publishers";
            try (ResultSet rs = stat.executeQuery(query)) {
               while (rs.next())
                  publishers.add(rs.getString(1));
            }
         }
         boolean done = false;
         while (!done) {
            String input = IO.readln("Q)uery C)hange prices E)xit: ").toUpperCase(
            if (input.equals("Q"))
               executeQuery(conn);
            else if (input.equals("C"))
               changePrices(conn);
            else
               done = true;
         }
      }
      catch (SQLException e) {
         for (Throwable t : e)
            IO.println(t.getMessage());
      }
   }

   /**
    * Executes the selected query.
    * @param conn the database connection
    */
   void executeQuery(Connection conn) throws SQLException {
      String author = select("Authors:", authors);
      String publisher = select("Publishers:", publishers);
      PreparedStatement stat;
      if (!author.equals("Any") && !publisher.equals("Any")) {
         stat = conn.prepareStatement(authorPublisherQuery);
         stat.setString(1, author);
         stat.setString(2, publisher);
      }
      else if (!author.equals("Any") && publisher.equals("Any")) {
         stat = conn.prepareStatement(authorQuery);
         stat.setString(1, author);
      }
      else if (author.equals("Any") && !publisher.equals("Any")) {
         stat = conn.prepareStatement(publisherQuery);
         stat.setString(1, publisher);
      }
```

```
      else
         stat = conn.prepareStatement(allQuery);

      try (ResultSet rs = stat.executeQuery()) {
         while (rs.next())
            IO.println(rs.getString(1) + ", " + rs.getString(2));
      }
   }

   /**
    * Executes an update statement to change prices.
    * @param conn the database connection
    */
   void changePrices(Connection conn) throws SQLException {
      String publisher = select("Publishers:", publishers.subList(1, publishers.siz
      double priceChange = Double.parseDouble(IO.readln("Change prices by: "
      PreparedStatement stat = conn.prepareStatement(priceUpdate);
      stat.setDouble(1, priceChange);
      stat.setString(2, publisher);
      int r = stat.executeUpdate();
      IO.println(r + " records updated.");
   }

   /**
    * Asks the user to select a string.
    * @param prompt the prompt to display
    * @param options the options from which the user can choose
    * @return the option that the user chose
    */
   String select(String prompt, List<String> options) {
      while (true) {
         IO.println(prompt);
         for (int i = 0; i < options.size(); i++)
            IO.println("%2d) %s".formatted(i + 1, options.get(i)));
         int sel = Integer.parseInt(IO.readln());
         if (sel > 0 && sel <= options.size()) return options.get(sel - 1);
      }
   }
}
```

java.sql.Connection **1.1**

- PreparedStatement prepareStatement(String sql)
  returns a PreparedStatement object containing the

precompiled statement. The string sql contains a SQL statement with one or more parameter placeholders denoted by ? characters.

java.sql.PreparedStatement1.1

- void set*Xxx*(int n, *Xxx* x)
  (*Xxx* is a type such as int, double, String, Date, etc.)
  sets the value of the nth parameter to x.
- void clearParameters()
  clears all current parameters in the prepared statement.
- ResultSet executeQuery()
  executes a prepared SQL query and returns a ResultSet object.
- int executeUpdate()
  executes the prepared SQL INSERT, UPDATE, or DELETE statement represented by the PreparedStatement object. Returns the number of rows affected, or 0 for DDL statements such as CREATE TABLE.

5.5.2. Reading and Writing LOBs

In addition to numbers, strings, and dates, many databases can store large objects (LOBs) such as images or other data. In SQL, binary large objects are called BLOBs, and character large objects are called CLOBs.

It depends on the application whether it is better to store large objects in the database or a file system. This section tells you what to do once you have decided to use a database for that purpose.

To read a LOB, execute a SELECT statement and call the getBlob or getClob method on the ResultSet. You will get an object of type Blob or Clob. To get the binary data from a Blob, call the getBytes or getBinaryStream. For example, if you have a table

with book cover images, you can retrieve an image like this:

```
PreparedStatement pstat
   = conn.prepareStatement("SELECT Cover FROM BookCovers WHE
. . .
pstat.set(1, isbn);
try (ResultSet result = pstat.executeQuery()) {
   if (result.next()) {
      Blob coverBlob = result.getBlob(1);
      Image coverImage = ImageIO.read(coverBlob.getBinaryStream()
   }
}
```

Similarly, if you retrieve a Clob object, you can get character data by calling the getSubString or getCharacterStream method.

To place a LOB into a database, call createBlob or createClob on your Connection object, get an output stream or writer to the LOB, write the data, and store the object in the database. For example, here is how you store an image:

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
ImageIO.write(coverImage, "PNG", out);
PreparedStatement pstat = conn.prepareStatement("INSERT INTO Co
pstat.set(1, isbn);
pstat.set(2, coverBlob);
pstat.executeUpdate();
```

java.sql.ResultSet 1.1

- Blob getBlob(int columnIndex) 1.2
- Blob getBlob(String columnLabel) 1.2
- Clob getClob(int columnIndex) 1.2
- Clob getClob(String columnLabel) 1.2

  get the BLOB or CLOB at the given column.

java.sql.Blob 1.2

- long length()
  gets the length of this BLOB.
- byte[] getBytes(long startPosition, long length)
  gets the data in the given range from this BLOB.
- InputStream getBinaryStream()
- InputStream getBinaryStream(long startPosition, long length)
  return a stream to read the data from this BLOB or from the given range.
- OutputStream setBinaryStream(long startPosition)1.4
  returns an output stream for writing into this BLOB, starting at the given position.

java.sql.Clob1.4

- long length()
  gets the number of characters of this CLOB.
- String getSubString(long startPosition, long length)
  gets the characters in the given range from this CLOB.
- Reader getCharacterStream()
- Reader getCharacterStream(long startPosition, long length)
  return a reader (not a stream) to read the characters from this CLOB or from the given range.
- Writer setCharacterStream(long startPosition)1.4
  returns a writer (not a stream) for writing into this CLOB, starting at the given position.

java.sql.Connection1.1

- Blob createBlob()6
- Clob createClob()6
  create an empty BLOB or CLOB.

5.5.3. SQL Escapes

The “escape” syntax features are commonly supported by databases but use database-specific syntax variations. It is the job of the JDBC driver to translate the escape syntax to the syntax of a particular database.

Escapes are provided for the following features:

- Date and time literals
- Calling scalar functions
- Calling stored procedures
- Outer joins
- The escape character in LIKE clauses

Date and time literals vary widely among databases. To embed a date or time literal, specify the value in one of the ISO 8601/ RFC 3339 formats shown below, where the milliseconds are optional. The driver will then translate it into the native format. Use d, t, ts for DATE, TIME, or TIMESTAMP values:

```
{d '2024-01-24'}
{t '23:59:59'}
{ts '2024-01-24 23:59:59.999'}
```

A scalar function is a function that returns a single value. Many functions are widely available in databases, but with varying names. The JDBC specification provides standard names and translates them into the database-specific names. To call a function, embed the standard function name and arguments like this:

```
{fn left(?, 20)}
{fn user()}
```

You can find a complete list of supported function names in the JDBC specification.

An outer join of two tables does not require that the rows of each table match according to the join condition. For example, the query

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers
```

```
ON Books.Publisher_Id = Publisher.Publisher_Id}
```

contains books for which Publisher_Id has no match in the Publishers table, with NULL values to indicate that no match exists. You would need a RIGHT OUTER JOIN to include publishers without matching books, or a FULL OUTER JOIN to return both. Most databases support all forms of outer joins, but some do not. The oj escape allows a JDBC driver to locate and rewrite any unsupported outer join operations. If your databases support all forms of outer joins, you need not worry about this escape.

Finally, the _ and % characters have special meanings in a LIKE clause—to match a single character or a sequence of characters. There is no standard way to use them literally. If you want to match all strings containing a _, use this construct:

```
. . . WHERE ? LIKE %!_% {escape '!'}
```

Here we define ! as the escape character. The combination !_ denotes a literal underscore.

A stored procedure is a procedure that executes in the database, written in a database-specific language. To invoke a stored procedure, use the call escape. You need not supply parentheses if the procedure has no parameters. Use = to capture a return value:

```
{call PROC1(?, ?)}
{call PROC2}
{call ? = PROC3(?)}
```

You need to use the CallableStatement interface to execute a stored procedure. Set all input parameters and specify the types of any outputs.

```
CallableStatement cstat = conn.prepareCall("{call PROC4(?, ?)}")
cstat.setInt(1, id);
cstat.registerOutParameter(2, java.sql.Types.VARCHAR);
cstat.execute();
String name = cstat.getString(2);
```

5.5.4. Multiple Results

It is possible for a query to return multiple result sets or update counts. This can happen when executing a stored procedure, or with databases that also allow submission of multiple SELECT statements in a single query. Here is how you retrieve all result sets:

1. Use the execute method to execute the SQL statement.
2. Retrieve the first result set or update count.
3. Repeatedly call the getMoreResults method to move on to the next result set or update count.
4. Finish when there are no more result sets or update counts.

The execute and getMoreResults methods return true if the next item in the chain is a result set. The getUpdateCount method returns -1 if the next item in the chain is not an update count.

The following loop traverses all results:

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done) {
   if (isResult) {
      ResultSet result = stat.getResultSet();
      do something with result
   }
   else {
      int updateCount = stat.getUpdateCount();
      if (updateCount >= 0)
         do something with updateCount
      else
         done = true;
   }
   if (!done) isResult = stat.getMoreResults();
}
```

java.sql.Statement1.1

- boolean getMoreResults()
- boolean getMoreResults(int current)6
  get the next result for this statement. The current parameter is one of CLOSE_CURRENT_RESULT (default), KEEP_CURRENT_RESULT, or CLOSE_ALL_RESULTS. Return true if the next result exists and is a result set.

5.5.5. Retrieving Autogenerated Keys

Most databases support some mechanism for autonumbering rows in a table. Unfortunately, the mechanisms differ widely among vendors. These automatic numbers are often used as primary keys. Although JDBC doesn't offer a vendor-independent solution for generating keys, it does provide an efficient way of retrieving them. When you insert a new row into a table and a key is automatically generated, you can retrieve it with the following code:

```
stat.executeUpdate(insertStatement, Statement.RETURN_GENERATE
ResultSet rs = stat.getGeneratedKeys();
if (rs.next()) {
   int key = rs.getInt(1);
   . . .
}
```

java.sql.Statement1.1

- boolean execute(String statement, int autogenerated)1.4
- int executeUpdate(String statement, int autogenerated)1.4
  execute the given SQL statement, as previously described. If autogenerated is set to Statement.RETURN_GENERATED_KEYS and the statement is an INSERT statement, the first column contains the autogenerated key.

5.6. Scrollable and Updatable Result Sets

As you have seen, the next method of the ResultSet interface iterates over the rows in a result set. That is certainly adequate for a program that needs to analyze the data. However, consider a visual data display that shows a table or query results (such as Figure 5.4). You usually want the user to be able to move both forward and backward in the result set. In a scrollable result, you can move forward and backward through a result set and even jump to any position.

Furthermore, once users see the contents of a result set displayed, they may be tempted to edit it. In an updatable result set, you can programmatically update entries so that the database is automatically updated. We discuss these capabilities in the following sections.

5.6.1. Scrollable Result Sets

By default, result sets are not scrollable or updatable. To obtain scrollable result sets from your queries, you must obtain a different Statement object with the method

```
Statement stat = conn.createStatement(type, concurrency);
```

For a prepared statement, use the call

```
PreparedStatement pstat = conn.prepareStatement(command, type,
```

The possible values of type and concurrency are listed in Table 5.6. You have the following choices:

- Do you want the result set to be scrollable? If not, use ResultSet.TYPE_FORWARD_ONLY.
- If the result set is scrollable, do you want it to reflect changes in the database that occurred after the query that yielded it? (In our discussion, we assume the ResultSet.TYPE_SCROLL_INSENSITIVE setting for scrollable result sets. This assumes that the result set does not “sense” database changes that occurred after the

execution of the query.)

- Do you want to be able to update the database by editing the result set? (See the next section for details.)

Value	Explanation
TYPE_FORWARD_ONLY	The result set is not scrollable (default).
TYPE_SCROLL_INSENSITIVE	The result set is scrollable but not sensitive to database changes.
TYPE_SCROLL_SENSITIVE	The result set is scrollable and sensitive to database changes.
CONCUR_READ_ONLY	The result set cannot be used to update the database (default).
CONCUR_UPDATABLE	The result set can be used to update the database.

Table 5.6: ResultSet Type and Concurrency Values

For example, if you simply want to be able to scroll through a result set but don't want to edit its data, use

```
Statement stat = conn.createStatement(
  ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_C
```

All result sets that are returned by calls

```
ResultSet rs = stat.executeQuery(query);
```

are now scrollable. A scrollable result set has a cursor that indicates the current position.

Not all database drivers support scrollable or updatable result sets. (The supportsResultSetType and supportsResultSetConcurrency methods of the DatabaseMetaData interface will tell you which types and concurrency modes are supported by a particular database using a particular driver.) Even if a database supports all result set modes, a particular query might not be able to yield a result set with all the properties that you requested. (For example, the result set of a complex query might not be updatable.) In that case, the executeQuery method returns a ResultSet of lesser

capabilities and adds a SQLWarning to the connection object. (Section 5.4.3 shows how to retrieve the warning.) Alternatively, you can use the getType and getConcurrency methods of the ResultSet interface to find out what mode a result set actually has. If you do not check the result set capabilities and issue an unsupported operation, such as previous on a result set that is not scrollable, the operation will throw a SQLException.

Scrolling is very simple. Use

```
if (rs.previous()) . . .;
```

to scroll backward. The method returns true if the cursor is positioned on an actual row, or false if it is now positioned before the first row.

You can move the cursor backward or forward by any number of rows with the call

```
rs.relative(n);
```

If *n* is positive, the cursor moves forward. If *n* is negative, it moves backward. If *n* is zero, the call has no effect. If you attempt to move the cursor outside the current set of rows, it is set to point either after the last row or before the first row, depending on the sign of *n*. Then, the method returns false and the cursor does not move. The method returns true if the cursor is positioned on an actual row.

Alternatively, you can set the cursor to a particular row number:

```
rs.absolute(n);
```

To get the current row number, call

```
int currentRow = rs.getRow();
```

The first row in the result set has number 1. If the return value is 0, the cursor is not currently on a row—it is either before the

first row or after the last row.

The convenience methods first, last, beforeFirst, and afterLast move the cursor to the first, to the last, before the first, or after the last position.

Finally, the methods isFirst, isLast, isBeforeFirst, and isAfterLast test whether the cursor is at one of these special positions.

Using a scrollable result set is very simple. The hard work of caching the query data is carried out behind the scenes by the database driver.

5.6.2. Updatable Result Sets

If you want to edit the data in the result set and have the changes automatically reflected in the database, create an updatable result set. Updatable result sets don't have to be scrollable, but if you present data to a user for editing, you usually want to allow scrolling as well.

To obtain updatable result sets, create a statement as follows:

```
Statement stat = conn.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATA
```

The result sets returned by a call to executeQuery are then updatable.

> Not all queries return updatable result sets. If your query is a join that involves multiple tables, the result might not be updatable. However, if your query involves only a single table or if it joins multiple tables by their primary keys, you should expect the result set to be updatable. Call the getConcurrency method of the ResultSet interface to find out for sure.

For example, suppose you want to raise the prices of some books, but you don't have a simple criterion for issuing an

UPDATE statement. Then, you can iterate through all books and update prices based on arbitrary conditions.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next()) {
   if (. . .) {
      double increase = . . .;
      double price = rs.getDouble("Price");
      rs.updateDouble("Price", price + increase);
      rs.updateRow(); // make sure to call updateRow after updating fi
   }
}
```

There are update*Xxx* methods for all data types that correspond to SQL types, such as updateDouble, updateString, and so on; specify the name or the number of the column (as with the get*Xxx* methods), then the new value for the field.

If you use the update*Xxx* method whose first parameter is the column number, be aware that this is the column number in the result set. It could well be different from the column number in the database.

The update*Xxx* method changes only the row values, not the database. When you are done with the field updates in a row, you must call the updateRow method. That method sends all updates in the current row to the database. If you move the cursor to another row without calling updateRow, this row's updates are discarded from the row set and never communicated to the database. You can also call the cancelRowUpdates method to cancel the updates to the current row.

The preceding example shows how to modify an existing row. If you want to add a new row to the database, first use the moveToInsertRow method to move the cursor to a special position, called the insert row. Then, build up a new row in the insert row position by issuing update*Xxx* instructions. When you are done, call the insertRow method to deliver the new row

to the database. When you are done inserting, call moveToCurrentRow to move the cursor back to the position before the call to moveToInsertRow. Here is an example:

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

Note that you cannot influence where the new data is added in the result set or the database.

If you don't specify a column value in the insert row, it is set to a SQL NULL. However, if the column has a NOT NULL constraint, an exception is thrown and the row is not inserted.

Finally, you can delete the row under the cursor:

```
rs.deleteRow();
```

The deleteRow method immediately removes the row from both the result set and the database.

The updateRow, insertRow, and deleteRow methods of the ResultSet interface give you the same power as executing UPDATE, INSERT, and DELETE SQL statements. However, Java programmers might find it more natural to manipulate the database contents through result sets than by constructing SQL statements.

> If you are not careful, you can write staggeringly inefficient code with updatable result sets. It is much more efficient to execute an UPDATE statement than to make a query and iterate through the result, changing data along the way. Updatable result sets make sense for interactive programs in which a user can make arbitrary changes, but for most programmatic changes, a SQL

UPDATE is more appropriate.

JDBC 2 delivered further enhancements to result sets, such as the capability to update a result set with the most recent data if the data have been modified by another concurrent database connection. JDBC 3 added yet another refinement, specifying the behavior of result sets when a transaction is committed. However, these advanced features are outside the scope of this introductory chapter. We refer you to the JDBC™ API Tutorial and Reference, Third Edition, by Maydene Fisher, Jon Ellis, and Jonathan Bruce (Addison-Wesley, 2003) and the JDBC specification for more information.

java.sql.Connection 1.1

- Statement createStatement(int type, int concurrency) 1.2
- PreparedStatement prepareStatement(String command, int type, int concurrency) 1.2
  create a statement or prepared statement that yields result sets with the given type and concurrency. The type parameter is of the constants TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE, and concurrency is one of the constants CONCUR_READ_ONLY or CONCUR_UPDATABLE, all defined in the ResultSet interface.

java.sql.ResultSet 1.1

- int getType() 1.2
  returns the type of this result set—one of TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE.
- int getConcurrency() 1.2
  returns the concurrency setting of this result set—one of CONCUR_READ_ONLY or CONCUR_UPDATABLE.
- boolean previous() 1.2

moves the cursor to the preceding row. Returns true if the cursor is positioned on a row, or false if the cursor is positioned before the first row.

- int getRow() 1.2
  gets the number of the current row. Rows are numbered starting with 1.
- boolean absolute(int r) 1.2
  moves the cursor to row r. Returns true if the cursor is positioned on a row.
- boolean relative(int d) 1.2
  moves the cursor by d rows. If d is negative, the cursor is moved backward. Returns true if the cursor is positioned on a row.
- boolean first() 1.2
- boolean last() 1.2
  move the cursor to the first or last row. Return true if the cursor is positioned on a row.
- void beforeFirst() 1.2
- void afterLast() 1.2
  move the cursor before the first or after the last row.
- boolean isFirst() 1.2
- boolean isLast() 1.2
  test whether the cursor is at the first or last row.
- boolean isBeforeFirst() 1.2
- boolean isAfterLast() 1.2
  test whether the cursor is before the first or after the last row.
- void moveToInsertRow() 1.2
  moves the cursor to the insert row. The insert row is a special row for inserting new data with the update*Xxx* and insertRow methods.
- void moveToCurrentRow() 1.2
  moves the cursor back from the insert row to the row that it occupied when the moveToInsertRow method was called.
- void insertRow() 1.2
  inserts the contents of the insert row into the database and the result set.
- void deleteRow() 1.2
  deletes the current row from the database and the result

set.
- void update*Xxx*(int column, *Xxx* data)1.2
- void update*Xxx*(String columnName, *Xxx* data)1.2
(*Xxx* is a type such as int, double, String, Date, etc.)
update a field in the current row of the result set.
- void updateRow()1.2
sends the current row updates to the database.
- void cancelRowUpdates()1.2
cancels the current row updates.

java.sql.DatabaseMetaData1.1

- boolean supportsResultSetType(int type)1.2
returns true if the database can support result sets of the given type; type is one of the constants TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE of the ResultSet interface.
- boolean supportsResultSetConcurrency(int type, int concurrency)1.2
returns true if the database can support result sets of the given combination of type and concurrency. The type parameter is one of the constants TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE, and concurrency is one of the constants CONCUR_READ_ONLY or CONCUR_UPDATABLE, all defined in the ResultSet interface.

5.7. Row Sets

Scrollable result sets are powerful, but they have a major drawback. You need to keep the database connection open during the entire user interaction. However, a user can walk away from the computer for a long time, leaving the connection occupied. That is not good—database connections are scarce resources. In this situation, use a row set. The RowSet interface extends the ResultSet interface, but row sets don't have to be tied to a database connection.

Row sets are also suitable if you need to move a query result to a different tier of a complex application, or to another device such as a cell phone. You would never want to move a result set—its data structures can be huge, and it is tethered to the database connection.

5.7.1. Constructing Row Sets

The javax.sql.rowset package provides the following interfaces that extend the RowSet interface:

- A CachedRowSet allows disconnected operation. We will discuss cached row sets in the following section.
- A WebRowSet is a cached row set that can be saved to an XML file. The XML file can be moved to another tier of a web application where it is opened by another WebRowSet object.
- The FilteredRowSet and JoinRowSet interfaces support lightweight operations on row sets that are equivalent to SQL SELECT and JOIN operations. These operations are carried out on the data stored in row sets, without having to make a database connection.
- A JdbcRowSet is a thin wrapper around a ResultSet. It adds useful methods from the RowSet interface.

To obtain a cached row set, call:

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

There are similar methods for obtaining the other row set types.

5.7.2. Cached Row Sets

A cached row set contains all data from a result set. Since CachedRowSet is a subinterface of the ResultSet interface, you can use a cached row set exactly as you would use a result set. Cached row sets confer an important benefit: You can close the connection and still use the row set. As you will see in our

sample program in Listing 5.4, this greatly simplifies the implementation of interactive applications. Each user command simply opens the database connection, issues a query, puts the result in a cached row set, and then closes the database connection.

It is even possible to modify the data in a cached row set. Of course, the modifications are not immediately reflected in the database; you need to make an explicit request to accept the accumulated changes. The CachedRowSet then reconnects to the database and issues SQL statements to write the accumulated changes.

You can populate a CachedRowSet from a result set:

```
ResultSet result = . . .;
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
crs.populate(result);
conn.close(); // now OK to close the database connection
```

Alternatively, you can let the CachedRowSet object establish a connection automatically. Set up the database parameters:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Then set the query statement and any parameters:

```
crs.setCommand("SELECT * FROM Books WHERE Publisher_Id = ?");
crs.setString(1, publisherId);
```

Finally, populate the row set with the query result:

```
crs.execute();
```

This call establishes a database connection, issues the query, populates the row set, and disconnects.

If your query result is very large, you would not want to put it into the row set in its entirety. After all, your users will

probably only look at a few rows. In that case, specify a page size:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

Now you will only get 20 rows. To get the next batch of rows, call

```
crs.nextPage();
```

You can inspect and modify the row set with the same methods you use for result sets. If you modified the row set contents, you must write it back to the database by calling

```
crs.acceptChanges(conn);
```

or

```
crs.acceptChanges();
```

The second call works only if you configured the row set with the information required to connect to a database (such as the URL, username, and password).

In Section 5.6.2, you saw that not all result sets are updatable. Similarly, a row set that contains the result of a complex query will not be able to write its changes back to the database. You should be safe if your row set contains data from a single table.

If you populated the row set from a result set, the row set does not know the name of the table to update. You need to call setTableName to set the table name.

Another complexity arises if the data in the database have changed after you populated the row set. This is clearly a sign of trouble that could lead to inconsistent data. The reference implementation checks whether the original row set values

(that is, the values before editing) are identical to the current values in the database. If so, they are replaced with the edited values; otherwise, a SyncProviderException is thrown and none of the changes are written. Other implementations may use other strategies for synchronization.

javax.sql.RowSet1.4

- String getURL()
- void setURL(String url)
  get or set the database URL.
- String getUsername()
- void setUsername(String username)
  get or set the username for connecting to the database.
- String getPassword()
- void setPassword(String password)
  get or set the password for connecting to the database.
- String getCommand()
- void setCommand(String command)
  get or set the command that is executed to populate this row set.
- void execute()
  populates this row set by issuing the statement set with setCommand. For the driver manager to obtain a connection, the URL, username, and password must be set.

javax.sql.rowset.CachedRowSet5.0

- void execute(Connection conn)
  populates this row set by issuing the statement set with setCommand. This method uses the given connection and closes it.
- void populate(ResultSet result)
  populates this cached row set with the data from the given result set.
- String getTableName()
- void setTableName(String tableName)
  get or set the name of the table from which this cached

row set was populated.
- int getPageSize()
- void setPageSize(int size)
  get or set the page size.
- boolean nextPage()
- boolean previousPage()
  load the next or previous page of rows. Return true if there is a next or previous page.
- void acceptChanges()
- void acceptChanges(Connection conn)
  reconnect to the database and write the changes that are the result of editing the row set. May throw a SyncProviderException if the data cannot be written back because the database data have changed.

javax.sql.rowset.RowSetProvider7

- static RowSetFactory newFactory()
  creates a row set factory.

javax.sql.rowset.RowSetFactory7

- CachedRowSet createCachedRowSet()
- FilteredRowSet createFilteredRowSet()
- JdbcRowSet createJdbcRowSet()
- JoinRowSet createJoinRowSet()
- WebRowSet createWebRowSet()
  create a row set of the specified type.

5.8. Metadata

In the preceding sections, you saw how to populate, query, and update database tables. However, JDBC can give you additional information about the structure of a database and its tables. For example, you can get a list of the tables in a particular

database or the column names and types of a table. This information is not useful when you are implementing a business application with a predefined database. After all, if you design the tables, you know their structure. Structural information is, however, extremely useful for programmers who write tools that work with any database.

In SQL, data that describe the database or one of its parts are called metadata (to distinguish them from the actual data stored in the database). You can get three kinds of metadata: about a database, about a result set, and about parameters of prepared statements.

To find out more about the database, request an object of type DatabaseMetaData from the database connection.

```
DatabaseMetaData meta = conn.getMetaData();
```

Use this object to get information about the database. For example, the call

```
ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE
```

returns a result set that contains information about all tables in the database. (See the API note at the end of this section for other parameters to this method.)

Each row in the result set contains information about a table in the database. The third column is the name of the table. (Again, see the API note for the other columns.) The following loop gathers all table names:

```
while (mrs.next())
   tableNames.add(mrs.getString(3));
```

There is a second important use for database metadata. Databases are complex, and the SQL standard leaves plenty of room for variability. Well over a hundred methods in the DatabaseMetaData interface can inquire about the database, including calls with such exotic names as

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

and

```
meta.nullPlusNonNullIsNull()
```

Clearly, these are geared toward advanced users with special needs—in particular, those who need to write highly portable code that works with multiple databases.

The DatabaseMetaData interface gives data about the database. A second metadata interface, ResultSetMetaData, reports information about a result set. Whenever you have a result set from a query, you can inquire about the number of columns and each column's name, type, and field width. Here is a typical loop:

```
ResultSet rs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData rsmd = rs.getMetaData();
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
   String columnName = rsmd.getColumnLabel(i);
   int columnWidth = rsmd.getColumnDisplaySize(i);
   . . .
}
```

If you have a prepared statement, you can get the metadata of the result set that is returned when the statement is executed:

```
PreparedStatement pstat = conn.prepareStatement(. . .);
ResultSetMetaData rsmd = pstat.getMetaData();
```

A third kind of metadata describes the parameters of a prepared statement:

```
ParameterMetaData pmd = pstat.getParameterMetaData();
for (int i = 1; i <= pmd.getParameterCount()) {
   IO.println(pmd.getParameterTypeName(i));
}
```

The program in Listing 5.4 uses metadata to let you browse all tables in a database and carry out simple edits. The program also illustrates the use of a cached row set.

The companion code has a prettier GUI version. The combo box

on top displays all tables in the database. Select one of them, and the center of the frame is filled with the field names of that table and the values of the first row, as shown in Figure 5.6. Click Next and Previous to scroll through the rows in the table. You can also delete a row and edit values of type String. Click the Save button to save the changes to the database.

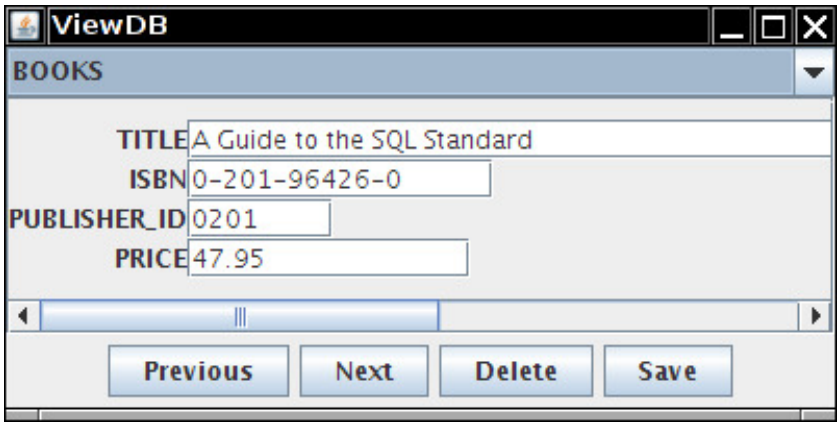


Figure 5.6: The ViewDB GUI application

The example program shows you how to implement tools for working with arbitrary tables. Many databases and IDEs come with much more sophisticated tools for viewing and editing tables. If you prefer a standalone program, check out DBeaver (https://dbeaver.io) or SQuirreL (https://squirrelsql.org). These programs can be used with any JDBC database.

Listing 5.4 **v2ch05/view/ViewDB2.java**

```
package v2ch05.view;

import module java.base;
import module java.sql;
import module java.sql.rowset;
```

```
import v2ch05.exec.ExecSQL;
import v2ch07.util.Choices;

/**
 * This program uses metadata to display arbitrary tables in a database.
 */
class ViewDB2 {
   CachedRowSet crs;

   record Column(String label, int width, String columnClass) {
   }

   List<Column> columns = new ArrayList<>();

   void main() {
      try (Connection conn = ExecSQL.getConnection()) {
         DatabaseMetaData meta = conn.getMetaData();
         var tableNames = new ArrayList<String>();

         try (ResultSet mrs = meta.getTables(null, null, null, new String[]{ "TABL
            while (mrs.next())
               tableNames.add(mrs.getString(3));
         }
         IO.println("Choose a table");
         String selected = Choices.choose(tableNames);
         showTable(selected, conn);
         executeCommands();
         crs.acceptChanges(conn);
      }
      catch (SQLException e) {
         for (Throwable t : e)
            t.printStackTrace();
      }
      catch (IOException e) {
         e.printStackTrace();
      }
   }

   /**
    * Prepares the columns list for showing a new table, and shows the first rov
    * @param tableName the name of the table to display
    * @param conn the database connection
    */
   void showTable(String tableName, Connection conn) throws SQLException {
```

```
   try (Statement stat = conn.createStatement();
        ResultSet result = stat.executeQuery("SELECT * FROM " + tableName)) {
      // copy into cached row set
      RowSetFactory factory = RowSetProvider.newFactory();
      crs = factory.createCachedRowSet();
      crs.setTableName(tableName);
      crs.populate(result);

      ResultSetMetaData rsmd = result.getMetaData();
      for (int i = 1; i <= rsmd.getColumnCount(); i++) {
         columns.add(new Column(rsmd.getColumnLabel(i), rsmd.getColumnl
               rsmd.getColumnClassName(i)));
      }

      showNextRow();
   }
}

void executeCommands() throws SQLException {
   while (true) {
      String selection = Choices.choose("Next", "Previous", "Edit", "Delete", "Qu
      switch (selection) {
         case "Next" -> showNextRow();
         case "Previous" -> showPreviousRow();
         case "Edit" -> editRow();
         case "Delete" -> deleteRow();
         default -> {
            return;
         }
      }
   }
}

/**
 * Shows a database row by populating all text fields with the column values
 */
void showRow(ResultSet rs) throws SQLException {
   for (int i = 1; i <= columns.size(); i++) {
      String format = "%%s [%%%ds]".formatted(columns.get(i - 1).width());
      IO.println(format.formatted(columns.get(i - 1).label(),
            rs == null ? "" : rs.getString(i)));
   }
}
```

```
   /**
    * Moves to the previous table row.
    */
   void showPreviousRow() throws SQLException {
      if (crs == null || crs.isFirst()) return;
      crs.previous();
      showRow(crs);
   }

   /**
    * Moves to the next table row.
    */
   void showNextRow() throws SQLException {
      if (crs == null || crs.isLast()) return;
      crs.next();
      showRow(crs);
   }

   /**
    * Deletes current table row.
    */
   void deleteRow() throws SQLException {
      if (crs == null) return;
      crs.deleteRow();
      if (crs.isAfterLast()) {
         if (!crs.last()) crs = null;
      }
      showRow(crs);
   }

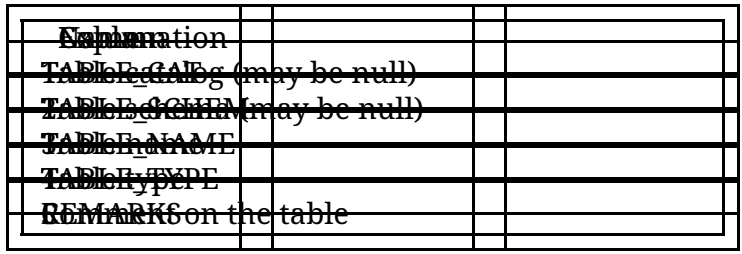
   /**
    * Updates changed data into the current row of the row set.
    */
   void editRow() throws SQLException {
      List<String> editableColumns = columns.stream()
            .filter(c -> c.columnClass.equals("java.lang.String")).map(Column::labe
            .toList();
      IO.println("Choose column");
      String label = Choices.choose(editableColumns);
      String newValue = IO.readln("New value: ");
      crs.updateString(label, newValue);
      crs.updateRow();
   }
}
```

java.sql.Connection1.1

- DatabaseMetaData getMetaData()
  returns the metadata for the connection as a DatabaseMetaData object.

java.sql.DatabaseMetaData1.1

- ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])
  returns a description of all tables in a catalog that match the schema and table name patterns and the type criteria. (A schema describes a group of related tables and access permissions. A catalog describes a related group of schemas. These concepts are important for structuring large databases.)
  The catalog and schemaPattern parameters can be "" to retrieve those tables without a catalog or schema, or null to return tables regardless of catalog or schema.
  The types array contains the names of the table types to include. Typical types are TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, and SYNONYM. If types is null, tables of all types are returned.
  The result set has five columns, all of which are of type String.



- int getJDBCMajorVersion()1.4
- int getJDBCMinorVersion()1.4
  return the major or minor JDBC version numbers of the driver that established the database connection. For example, a JDBC 4.3 driver has major version number 4 and minor version number 3.

- int getMaxConnections()
  returns the maximum number of concurrent connections allowed to this database.
- int getMaxStatements()
  returns the maximum number of concurrently open statements allowed per database connection, or 0 if the number is unlimited or unknown.

java.sql.ResultSet1.1

- ResultSetMetaData getMetaData()
  returns the metadata associated with the current ResultSet columns.

java.sql.PreparedStatement1.1

- ResultSetMetaData getMetaData()1.2
  returns the metadata associated with the result set that is returned when this statement is executed.
- ParameterMetaData getMetaData()1.4
  returns metadata describing the parameters of this statement.

java.sql.ResultSetMetaData1.1

- int getColumnCount()
  returns the number of columns in the current ResultSet object.
- String getColumnName(int column)
- String getColumnLabel(int column)
- int getColumnDisplaySize(int column)
  return the name, suggested title, and display size of the column with the given one-based index.

java.sql.ParameterMetaData1.4

- int getParameterCount()
  returns the number of parameters of the prepared statement.
- int getParameterType(int param)
- String getParameterTypeName(int param)
- String getParameterClassName(int param)
  return the type of the parameter with the given one-based index, as a constant defined in java.sql.Types, or as a database-specific type name, or as a fully-qualified Java class name.
- int getParameterMode(int param)
  returns the mode of the parameter with the given one-based index, as one of the static constants parameterModeIn, parameterModeOut, parameterModeInOut, or parameterModeUnknown.
- int isNullable(int param)
  returns the nullability mode of the parameter with the given one-based index, as one of the static constants parameterNoNulls, parameterNullable, or parameterNullableUnknown.
- int getPrecision(int param)
  returns the maximum column size of the parameter with the given one-based index. For non-numeric data types, this is the length of the string or string representation, or the number of bytes for binary data.
- int getScale(int param)
  returns the number of digits after the decimal point of the parameter with the given one-based index, or 0 for non-numeric data.

5.9. Transactions

You can group a set of statements to form a transaction. The transaction can be committed when all has gone well—or, if an error has occurred in one of them, it can be rolled back as if none of the statements had been issued.

The major reason for grouping statements into transactions is

database integrity. For example, suppose we want to transfer money from one bank account to another. Then, it is important that we simultaneously debit one account and credit another. If the system fails after debiting the first account but before crediting the other account, the debit needs to be undone.

If you group update statements into a transaction, the transaction either succeeds in its entirety and can be committed, or it fails somewhere in the middle. In that case, you can carry out a rollback and the database automatically undoes the effect of all updates that occurred since the last committed transaction.

5.9.1. Programming Transactions with JDBC

By default, a database connection is in autocommit mode, and each SQL statement is committed to the database as soon as it is executed. Once a statement is committed, you cannot roll it back. Turn off this default so you can use transactions:

```
conn.setAutoCommit(false);
```

Create a statement object in the normal way:

```
Statement stat = conn.createStatement();
```

Call executeUpdate any number of times:

```
stat.executeUpdate(command1);
stat.executeUpdate(command2);
stat.executeUpdate(command3);
. . .
```

If all statements have been executed without error, call the commit method:

```
conn.commit();
```

However, if an error occurred, call

```
conn.rollback();
```

Then, all statements since the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a SQLException.

5.9.2. Save Points

With some databases and drivers, you can gain finer-grained control over the rollback process by using save points. Creating a save point marks a point to which you can later return without having to abandon the entire transaction. For example,

```
Statement stat = conn.createStatement(); // start transaction; rollback
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint(); // set savepoint; rollback(svpt)
stat.executeUpdate(command2);

if (. . .) conn.rollback(svpt); // undo effect of command2
. . .
conn.commit();
```

When you no longer need a save point, you should release it:

```
conn.releaseSavepoint(svpt);
```

5.9.3. Batch Updates

Suppose a program needs to execute many INSERT statements to populate a database table. You can improve the performance of the program by using a batch update. In a batch update, a sequence of statements is collected and submitted as a batch.

Use the supportsBatchUpdates method of the DatabaseMetaData interface to find out if your database supports this feature.

The statements in a batch can be actions such as INSERT, UPDATE, or DELETE as well as data definition statements such as CREATE TABLE or DROP TABLE. An exception is thrown if you add a SELECT statement to a batch. (Conceptually, a

SELECT statement makes no sense in a batch because it returns a result set without updating the database.)

To execute a batch, first create a Statement object in the usual way:

```
Statement stat = conn.createStatement();
```

Now, instead of calling executeUpdate, call the addBatch method:

```
String command = "CREATE TABLE . . ."
stat.addBatch(command);

while (. . .) {
   command = "INSERT INTO . . . VALUES (" + . . . + ")";
   stat.addBatch(command);
}
```

Finally, submit the entire batch:

```
int[] counts = stat.executeBatch();
```

The call to executeBatch returns an array of the row counts for all submitted statements.

For proper error handling in batch mode, treat the batch execution as a single transaction. If a batch fails in the middle, you want to roll back to the state before the beginning of the batch.

First, turn the autocommit mode off, then collect the batch, execute it, commit it, and finally restore the original autocommit mode:

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.getStatement();
. . .
// keep calling stat.addBatch(. . .);
. . .
stat.executeBatch();
```

conn.commit(); conn.setAutoCommit(autoCommit);

java.sql.Connection1.1

- boolean getAutoCommit()
- void setAutoCommit(boolean b)
  get or set the autocommit mode of this connection to b. If autocommit is true, all statements are committed as soon as their execution is completed.
- void commit()
  commits all statements that were issued since the last commit.
- void rollback()
  undoes the effect of all statements that were issued since the last commit.
- Savepoint setSavepoint()1.4
- Savepoint setSavepoint(String name)1.4
  set an unnamed or named save point.
- void rollback(Savepoint svpt)1.4
  rolls back until the given save point.
- void releaseSavepoint(Savepoint svpt)1.4
  releases the given save point.

java.sql.Savepoint1.4

- int getSavepointId()
  gets the ID of this unnamed save point, or throws a SQLException if this is a named save point.
- String getSavepointName()
  gets the name of this save point, or throws a SQLException if this is an unnamed save point.

java.sql.Statement1.1

- void addBatch(String command)1.2
  adds the command to the current batch of commands for

this statement.

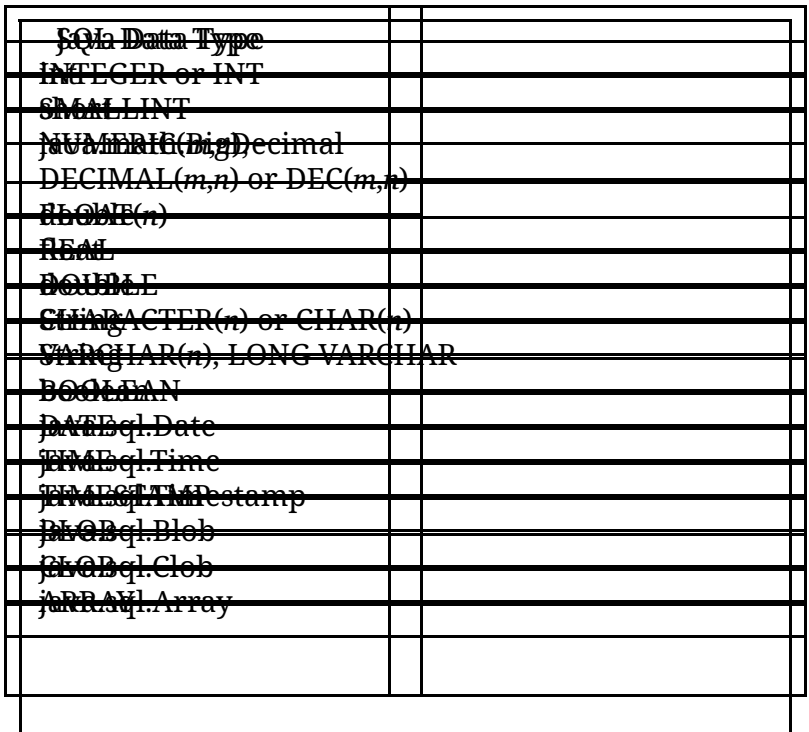
- int[] executeBatch()1.2
- long[] executeLargeBatch()8
  execute all commands in the current batch. Each value in the returned array corresponds to one of the batch statements. If it is non-negative, it is a row count. If it is the value SUCCESS_NO_INFO, the statement succeeded, but no row count is available. If it is EXECUTE_FAILED, the statement failed.

java.sql.DatabaseMetaData1.1

- boolean supportsBatchUpdates()1.2
  returns true if the driver supports batch updates.

5.9.4. Advanced SQL Types

Table 5.8 lists the SQL data types supported by JDBC and their equivalents in the Java programming language.



ROWID	java.sql.RowId
NCHAR(*n*), NVARCHAR(*n*), LONG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

Table 5.8: SQL Data Types and Their Corresponding Java Types

A SQL ARRAY is a sequence of values. For example, in a Student table, you can have a Scores column that is an ARRAY OF INTEGER. The getArray method returns an object of the interface type java.sql.Array. That interface has methods to fetch the array values.

When you get a LOB or an array from a database, the actual contents are fetched from the database only when you request individual values. This is a useful performance enhancement, as the data can be quite voluminous.

Some databases support ROWID values that describe the location of a row so that it can be retrieved very rapidly. JDBC 4 introduced an interface java.sql.RowId and the methods to supply the row ID in queries and retrieve it from results.

A national character string (NCHAR and its variants) stores strings in a local character encoding and sorts them using a local sorting convention. JDBC 4 provided methods for converting between Java String objects and national character strings in queries and results.

Some databases can store user-defined structured types. JDBC 3 provides a mechanism for automatically mapping structured SQL types to Java objects.

Some databases provide native storage for XML data. JDBC 4 introduced a SQLXML interface that can mediate between the internal XML representation and the DOM Source/Result interfaces, as well as binary streams. See the API documentation for the SQLXML class for details.

We do not discuss these advanced SQL types any further. You can find more information on these topics in the JDBC API

Tutorial and Reference and the JDBC specification.

5.10. Connection Management in Web and Enterprise Applications

The simplistic database connection setup with a database.properties file, as described in the preceding sections, is suitable for small test programs but won't scale for larger applications.

When a JDBC application is deployed in a web or enterprise environment, the management of database connections is integrated with a directory service called JNDI (Java Naming and Directory Interface). The properties of data sources across the enterprise can be stored in a directory. Using a directory allows for centralized management of usernames, passwords, database names, and JDBC URLs.

In such an environment, you can use the following code to establish a database connection:

```
var jndiContext = new InitialContext();
var source = (DataSource) jndiContext.lookup("java:comp/env/jdbc/c
Connection conn = source.getConnection();
```

Note that the DriverManager is no longer involved. Instead, the JNDI service locates a data source. A data source is an interface that allows for simple JDBC connections as well as more advanced services, such as executing distributed transactions that involve multiple databases. The DataSource interface is defined in the javax.sql standard extension package.

In a Java EE container, you don't even have to program the JNDI lookup. Simply use the Resource annotation on a DataSource field, and the data source reference will be set when your application is loaded:

```
@Resource(name="jdbc/corejava")
private DataSource source;
```

Of course, the data source needs to be configured somewhere. If you write database programs that execute in a servlet container such as Apache Tomcat or in an application server such as GlassFish, place the database configuration (including the JNDI name, JDBC URL, username, and password) in a configuration file, or set it in an admin GUI.

Management of usernames and logins is just one of the issues that require special attention. Another issue involves the cost of establishing database connections. Our sample database programs used two strategies for obtaining a database connection. The QueryDB program in Listing 5.3 established a single database connection at the start of the program and closed it at the end of the program. The ViewDB program in Listing 5.4 opened a new connection whenever one was needed.

However, neither of these approaches is satisfactory. Database connections are a finite resource. If a user walks away from an application for some time, the connection should not be left open. Conversely, obtaining a connection for each query and closing it afterward is very costly.

The solution is to pool connections. This means that database connections are not physically closed but are kept in a queue and reused. Connection pooling is an important service, and the JDBC specification provides hooks for implementors to supply it. However, the JDK itself does not provide any implementation, and database vendors don't usually include one with their JDBC drivers either. Instead, vendors of web containers and application servers supply connection pool implementations.

Using a connection pool is completely transparent to the programmer. Acquire a connection from a source of pooled connections by obtaining a data source and calling getConnection. When you are done using the connection, call close. That doesn't close the physical connection but tells the pool that you are done using it. The connection pool typically makes an effort to pool prepared statements as well.

You have now learned about the JDBC fundamentals and know enough to implement simple database applications. However, as we mentioned at the beginning of this chapter, databases are complex and quite a few advanced topics are beyond the scope of this introductory chapter. For an overview of advanced JDBC capabilities, refer to the JDBC API Tutorial and Reference or the JDBC specification.

In this chapter, you have learned how to work with relational databases in Java. The next chapter covers the date and time library.

Chapter 6 ▪ The Date and Time API

Time flies like an arrow, and we can easily set a starting point and count forward and backward in seconds. So why is it so hard to deal with time? The problem is humans. All would be easy if we could just tell each other: “Meet me at 1523793600, and don’t be late!” But we want time to relate to daylight and the seasons. That’s where things get complicated. Java 1.0 had a Date class that was, in hindsight, naïve, and had most of its methods deprecated in Java 1.1 when a Calendar class was introduced. Its API wasn’t stellar, its instances were mutable, and it didn’t deal with issues such as leap seconds. The third time is a charm, and the java.time API introduced in Java 8 has remedied the flaws of the past and should serve us for quite some time. In this chapter, you will learn what makes time computations so vexing and how the Date and Time API solves these issues.

6.1. The Time Line

Historically, the fundamental time unit—the second—was derived from Earth's rotation around its axis. There are 24 hours or 24 × 60 × 60 = 86400 seconds in a full revolution, so it seems just a question of astronomical measurements to precisely define a second. Unfortunately, Earth wobbles

slightly, and a more precise definition was needed. In 1967, a new precise definition of a second, matching the historical definition, was derived from an intrinsic property of atoms of caesium-133. Since then, a network of atomic clocks keeps the official time.

Ever so often, the official timekeepers synchronize the absolute time with the rotation of Earth. At first, the official seconds were slightly adjusted, but starting in 1972, “leap seconds” were occasionally inserted. (In theory, a second might need to be removed once in a while, but that has not yet happened.) There is talk of changing the system again. Clearly, leap seconds are a pain, and many computer systems instead use “smoothing” where time is artificially slowed down or sped up just before the leap second, keeping 86,400 seconds per day. This works because the local time on a computer isn’t all that precise, and computers are used to synchronizing themselves with an external time service.

The Java Date and Time API specification requires that Java uses a time scale that

- Has 86,400 seconds per day
- Exactly matches the official time at noon each day
- Closely matches it elsewhere, in a precisely defined way

That gives Java the flexibility to adjust to future changes in the official time.

In Java, an Instant represents a point on the time line. An origin, called the epoch, is arbitrarily set at midnight of January 1, 1970 at the prime meridian that passes through the Greenwich Royal Observatory in London. This is the same convention used in the UNIX/POSIX time. Starting from that origin, time is measured in 86,400 seconds per day, forward and backward, to nanosecond precision. The Instant values go back as far as a billion years (Instant.MIN). That's not quite enough to express the age of the universe (around 13.5 billion years) but it should be enough for all practical purposes. After all, a billion years ago, the earth was covered in ice and populated by microscopic ancestors of today’s plants and

animals. The largest value, Instant.MAX, is December 31 of the year 1,000,000,000.

The static method call Instant.now() gives the current instant. You can compare two instants with the equals and compareTo methods in the usual way, so you can use instants as timestamps.

To find out the difference between two instants, use the static method Duration.between. For example, here is how you can measure the running time of an algorithm:

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

A Duration is the amount of time between two instants. You can get the length of a Duration in conventional units by calling toNanos, toMillis, toSeconds, toMinutes, toHours, or toDays.

In Java 8, you had to call getSeconds instead of toSeconds.

If you need nanosecond precision, be aware of overflow. A long value can hold almost 300 years of nanoseconds. If your durations are shorter than that, simply convert them to nanoseconds. You can use longer durations—a Duration object stores the number of seconds in a long, and the number of nanoseconds in an additional int. The Duration API has a number of methods, shown at the end of this section, for carrying out arithmetic.

For example, if you want to check whether an algorithm is at least ten times faster than another, you can compute

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster
   = timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative()
```

This is just to show the syntax. Since the algorithms aren't going to run for hundreds of years, you can simply use

```
boolean overTenTimesFaster = timeElapsed.toNanos() * 10 < timeEla
```

> The Instant and Duration classes are immutable, and all methods, such as multipliedBy or minus, return a new instance.

The example program in Listing 6.1 uses the Instant and Duration classes for timing two algorithms.

> The approach of the example program is too naïve in practice, since it does not deal with issues such as the "warming up" of the just-in-time compiler in the JVM. If you need to compare algorithms, you should use a dedicated benchmarking tool such as the Java Microbenchmark Harness (JMH). The article at https://www.oracle.com/technical-resources/articles/java/architect-benchmarking.html gives useful advice.

Listing 6.1 **v2ch06/Timeline.java**

```
package v2ch06;

/**
 * This program demonstrates instants and durations.
 */

import module java.base;

public class Timeline {
   RandomGenerator generator = RandomGenerator.getDefault();

   void main() {
      Instant start = Instant.now();
      runAlgorithm();
      Instant end = Instant.now();
```

```
      Duration timeElapsed = Duration.between(start, end);
      long millis = timeElapsed.toMillis();
      IO.println("%d milliseconds".formatted(millis));

      Instant start2 = Instant.now();
      runAlgorithm2();
      Instant end2 = Instant.now();
      Duration timeElapsed2 = Duration.between(start2, end2);
      IO.println("%d milliseconds".formatted(timeElapsed2.toMillis()));
      boolean overTenTimesFaster = timeElapsed.multipliedBy(10).minus(timeE
            .isNegative();
      IO.println("The first algorithm is %smore than ten times faster".formatted
            overTenTimesFaster ? "" : "not "));
   }

   void runAlgorithm() {
      int size = 10;
      ArrayList<Integer> list = generator.ints().map(i -> i % 100).limit(size).boxed
            .collect(Collectors.toCollection(ArrayList::new));
      Collections.sort(list);
      IO.println(list);
   }

   void runAlgorithm2() {
      int size = 10;
      List<Integer> list = generator.ints().map(i -> i % 100).limit(size).boxed()
            .collect(Collectors.toCollection(ArrayList::new));
      while (!IntStream.range(1, list.size())
            .allMatch(i -> list.get(i - 1).compareTo(list.get(i)) <= 0))
         Collections.shuffle(list);
      IO.println(list);
   }
}
```

java.time.Instant8

- static Instant now()
  gets the current instant from the best available system clock.
- Instant plus(TemporalAmount amountToAdd)
- Instant minus(TemporalAmount amountToSubtract)
  yield an instant that is the given amount away from this Instant. The classes Duration and Period (see) implement

the TemporalAmount interface.

- Instant plus*Xxx*(long number)
- Instant minus*Xxx*(long number)
  yields an Instant that is the given number of time units away from this Instant, where *Xxx* is Nanos, Millis, or Seconds

java.time.Duration8

- static Duration of*Xxx*(long number)
  yields a duration of the given number of time units, where *Xxx* is Nanos, Millis, Seconds, Minutes, Hours, or Days.
- static Duration between(Temporal startInclusive, Temporal endExclusive)
  yields a duration between the given points in time. The Temporal interface is implemented by the Instant class as well as LocalDate/LocalDateTime/LocalTime (see) and ZonedDateTime (see).
- long to*Xxx*()
- long toSeconds()9
  get the number of the time units in the method name for this Duration, where *Xxx* is Nanos, Millis, Minutes, Hours, or Days.
- int to*Xxx*Part()9
- long toDaysPart()9
  yield the part of the given time unit in this Duration. For example, in a duration of 100 seconds, the minutes part is 1 and the seconds part is 40. Here, var Xxx is one of Nanos, Millis, Seconds, Minutes, or Hours.
- Duration plus(TemporalAmount amountToAdd)
- Duration minus(TemporalAmount amountToSubtract)
  yield a duration that is the given amount away from this Duration. The classes Duration and Period (see) implement the TemporalAmount interface.
- Duration multipliedBy(long multiplicand)
- Duration dividedBy(long divisor)
- Duration negated()
  yield a duration that is obtained by multiplying or dividing this Duration by the given amount, or by –1.

- boolean isZero()
- boolean isNegative()
  return true if this Duration is zero or negative.
- Duration plus*Xxx*(long number)
- Duration minus*Xxx*(long number)
  yields a Duration obtained by adding or subtracting the given number of time units, where *Xxx* is one of Nanos, Millis, Seconds, Minutes, or Hours.

6.2. Local Dates

Now let us turn from absolute time to human time. There are two kinds of human time in the Java API, local date/time and zoned time. Local date/time has a date and/or time of day, but no associated time zone information. An example of a local date is June 14, 1903 (the day on which Alonzo Church, inventor of the lambda calculus, was born). Since that date has neither a time of day nor time zone information, it does not correspond to a precise instant of time. In contrast, July 16, 1969, 09:32:00 EDT (the launch of Apollo 11) is a zoned date/time, representing a precise instant on the time line.

There are many calculations where time zones are not required, and in some cases they can even be a hindrance. Suppose you schedule a meeting every week at 10:00. If you add 7 days (that is, 7 × 24 × 60 × 60 seconds) to the last zoned time, and you happen to cross the daylight savings time boundary, the meeting will be an hour too early or too late!

For that reason, the API designers recommend that you do not use zoned time unless you really want to represent absolute time instances. Birthdays, holidays, schedule times, and so on are usually best represented as local dates or times.

A LocalDate is a date with a year, month, and day of the month. To construct one, you can use the now or of static methods:

```
LocalDate today = LocalDate.now(); // Today's date
LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
```

```
alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
   // Uses the Month enumeration
```

Unlike the irregular conventions in UNIX and java.util.Date, where months are zero-based and years are counted from 1900, here you supply the usual numbers for the month of year. Alternatively, you can use the Month enumeration.

The API notes at the end of this section show the most useful methods for working with LocalDate objects.

For example, Programmer's Day is the 256th day of the year. Here is how you can easily compute it:

```
LocalDate programmersDay = LocalDate.of(2014, 1, 1).plusDays(255)
   // September 13, but in a leap year it would be September 12
```

Recall that the difference between two time instants is a Duration. The equivalent for local dates is a Period, which expresses a number of elapsed years, months, or days. You can call birthday.plus(Period.ofYears(1)) to get the birthday next year. Of course, you can also just call birthday.plusYears(1). But birthday.plus(Period.ofDays(365)) won't produce the correct result in a leap year.

The until method yields the difference between two local dates. For example,

```
independenceDay.until(christmas)
```

yields a period of 5 months and 21 days. That is actually not terribly useful because the number of days per month varies. To find the number of days, use

```
independenceDay.until(christmas, ChronoUnit.DAYS) // 174 days
```

> Some methods in the LocalDate API could potentially create nonexistent dates. For example, adding one month to January 31 should not yield February 31. Instead of throwing an exception, these methods return the last valid day of the month. For example,

```
LocalDate.of(2016, 1, 31).plusMonths(1)
```

and

```
LocalDate.of(2016, 3, 31).minusMonths(1)
```

yield February 29, 2016.

The getDayOfWeek yields the weekday, as a value of the DayOfWeek enumeration. DayOfWeek.MONDAY has the numerical value 1, and DayOfWeek.SUNDAY has the value 7. For example,

```
LocalDate.of(1900, 1, 1).getDayOfWeek().getValue()
```

yields 1. The DayOfWeek enumeration has convenience methods plus and minus to compute weekdays modulo 7. For example, DayOfWeek.SATURDAY.plus(3) yields DayOfWeek.TUESDAY.

> The weekend days actually come at the end of the week. This is different from java.util.Calendar where Sunday has value 1 and Saturday value 7.

The useful datesUntil methods yield streams of LocalDate objects:

```
LocalDate start = LocalDate.of(2000, 1, 1);
LocalDate endExclusive = LocalDate.now();
Stream<LocalDate> allDays = start.datesUntil(endExclusive);
Stream<LocalDate> firstDaysInMonth = start.datesUntil(endExclusiv
```

In addition to LocalDate, there are also classes MonthDay, YearMonth, and Year to describe partial dates. For example, December 25 (with the year unspecified) can be represented as a MonthDay.

The example program in Listing 6.2 shows how to work with the LocalDate class.

Listing 6.2 **v2ch06/LocalDates.java**

```
package v2ch06;

/**
 * This program demonstrates local dates.
 */
import module java.base;

class LocalDates {
   void main() {
      LocalDate today = LocalDate.now(); // Today's date
      IO.println("today: " + today);

      LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
      alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
      // Uses the Month enumeration
      IO.println("alonzosBirthday: " + alonzosBirthday);

      LocalDate programmersDay = LocalDate.of(2018, 1, 1).plusDays(255);
      // September 13, but in a leap year it would be September 12
      IO.println("programmersDay: " + programmersDay);

      LocalDate independenceDay = LocalDate.of(2018, Month.JULY, 4);
      LocalDate christmas = LocalDate.of(2018, Month.DECEMBER, 25);

      IO.println("Until christmas: " + independenceDay.until(christmas));
      IO.println("Until christmas: " + independenceDay.until(christmas, Chronol

      IO.println(LocalDate.of(2016, 1, 31).plusMonths(1));
      IO.println(LocalDate.of(2016, 3, 31).minusMonths(1));

      DayOfWeek startOfLastMillennium = LocalDate.of(1900, 1, 1).getDayOfWe
      IO.println("startOfLastMillennium: " + startOfLastMillennium);
      IO.println(startOfLastMillennium.getValue());
      IO.println(DayOfWeek.SATURDAY.plus(3));

      LocalDate start = LocalDate.of(2000, 1, 1);
      LocalDate endExclusive = LocalDate.now();
      Stream<LocalDate> firstDaysInMonth = start.datesUntil(endExclusive,
            Period.ofMonths(1));
      IO.println("firstDaysInMonth: " + firstDaysInMonth.toList());
   }
}
```

java.time.LocalDate8

- static LocalDate now()
  gets the current LocalDate.
- static LocalDate of(int year, int month, int dayOfMonth)
- static LocalDate of(int year, Month month, int dayOfMonth)
  yield a local date with the given year, month (as integer between 1 and 12 or a value of the Month enumeration), and day of the month (between 1 and 31).
- LocalDate plus*Xxx*(long number)
- LocalDate minus*Xxx*(long number)
  yields a LocalDate obtained by adding or subtracting the given number of time units, where *Xxx* is one of Days, Weeks, Months, orYears.
- LocalDate plus(TemporalAmount amountToAdd)
- LocalDate minus(TemporalAmount amountToSubtract)
  yield a LocalDate that is the given amount away from this LocalDate. The classes Duration and Period implement the TemporalAmount interface.
- LocalDate withDayOfMonth(int dayOfMonth)
- LocalDate withDayOfYear(int dayOfYear)
- LocalDate withMonth(int month)
- LocalDate withYear(int year)
  return a new LocalDate with the day of month, day of year, month, or year changed to the given value.
- int getDayOfMonth()
  gets the day of the month (between 1 and 31).
- int getDayOfYear()
  gets the day of the year (between 1 and 366).
- DayOfWeek getDayOfWeek()
  gets the day of the week, returning a value of the DayOfWeek enumeration.
- Month getMonth()
- int getMonthValue()
  get the month as a value of the Month enumeration, or as a number between 1 and 12.
- int getYear()
  gets the year, between –999,999,999 and 999,999,999.
- Period until(ChronoLocalDate endDateExclusive)

gets the period until the given end date. The ChronoLocalDate interface is implemented by LocalDate and date classes for non-Gregorian calendars.

- boolean isBefore(ChronoLocalDate other)
- boolean isAfter(ChronoLocalDate other)
  return true if this date is before or after the given date.
- boolean isLeapYear()
  returns true if the year is a leap year—that is, if it is divisible by 4 but not by 100, or divisible by 400. The algorithm is applied for all past years, even though that is historically inaccurate. (Leap years were invented in the year –46, and the rules involving divisibility by 100 and 400 were introduced in the Gregorian calendar reform of 1582. The reform took over 300 years to become universal.)
- Stream<LocalDate> datesUntil(LocalDate endExclusive)9
- Stream<LocalDate> datesUntil(LocalDate endExclusive, Period step)9
  yield a stream of dates from this LocalDate until the end, with step size 1 or the given period.

java.time.Period8

- static Period of(int years, int months, int days)
- Period of*Xxx*(int number)
  yield a Period with the given number of time units, where *Xxx* is Days, Weeks, Months, or Years.
- int get*Xxx*()
  gets the number of time units of this Period, where *Xxx* is Days,Months, or Years.
- Period plus*Xxx*(long number)
- Period minus*Xxx*(long number)
  yields a Period obtained by adding or subtracting the given number of time units, where *Xxx* is Days,Months, or Years.
- Period plus(TemporalAmount amountToAdd)
- Period minus(TemporalAmount amountToSubtract)
  yield an instant that is the given amount away from this Instant. The classes Duration and Period implement the

TemporalAmount interface.

- Period with*Xxx*(int number)
  returns a new Period with the days, months, or years changed to the given number, where *Xxx* is Days,Months, or Years.

6.3. Date Adjusters

For scheduling applications, you often need to compute dates such as “the first Tuesday of every month.” The TemporalAdjusters class provides a number of static methods for common adjustments. You pass the result of an adjustment method to the with method. For example, the first Tuesday of a month can be computed like this:

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(
   TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

As always, the with method returns a new LocalDate object without modifying the original. The API notes at the end of this section show the available adjusters.

You can also make your own adjuster by implementing the TemporalAdjuster interface. Here is an adjuster for computing the next weekday:

```
TemporalAdjuster NEXT_WORKDAY = w -> {
   var result = (LocalDate) w;
   do {
      result = result.plusDays(1);
   }
   while (result.getDayOfWeek().getValue() >= 6);
   return result;
};

LocalDate backToWork = today.with(NEXT_WORKDAY);
```

Note that the parameter of the lambda expression has type Temporal, and it must be cast to LocalDate. You can avoid this

cast with the ofDateAdjuster method that expects a lambda of type UnaryOperator<LocalDate>.

```
TemporalAdjuster NEXT_WORKDAY = TemporalAdjusters.ofDateAdju
   LocalDate result = w; // No cast
   do {
      result = result.plusDays(1);
   }
   while (result.getDayOfWeek().getValue() >= 6);
   return result;
});
```

java.time.LocalDate9

- LocalDate with(TemporalAdjuster adjuster)
  returns the result of adjusting this date by the given adjuster.

java.time.temporal.TemporalAdjusters9

- static TemporalAdjuster next(DayOfWeek dayOfWeek)
- static TemporalAdjuster nextOrSame(DayOfWeek dayOfWeek)
- static TemporalAdjuster previous(DayOfWeek dayOfWeek)
- static TemporalAdjuster previousOrSame(DayOfWeek dayOfWeek)
  return an adjuster that adjusts a date to the given day of the week.
- static TemporalAdjuster dayOfWeekInMonth(int n, DayOfWeek dayOfWeek)
- static TemporalAdjuster lastInMonth(DayOfWeek dayOfWeek)
  return an adjuster that adjusts a date to the nth or last given weekday of the month.
- static TemporalAdjuster firstDayOfMonth()
- static TemporalAdjuster firstDayOfNextMonth()
- static TemporalAdjuster firstDayOfYear()
- static TemporalAdjuster firstDayOfNextYear()

- static TemporalAdjuster lastDayOfMonth()
- static TemporalAdjuster lastDayOfYear()
  return an adjuster that adjusts a date to the given day of the month or year.

6.4. Local Time

A LocalTime represents a time of day, such as 15:30:00. You can create an instance with the now or of methods:

```
LocalTime rightNow = LocalTime.now();
LocalTime bedtime = LocalTime.of(22, 30); // or LocalTime.of(22, 30,
```

The API notes show common operations with local times. The plus and minus operations wrap around a 24-hour day. For example,

```
LocalTime wakeup = bedtime.plusHours(8); // wakeup is 6:30:00
```

LocalTime doesn't concern itself with AM/PM. That silliness is left to a formatter—see Section 6.6.

There is a LocalDateTime class representing a date and time. That class is suitable for storing points in time in a fixed time zone—for example, for a schedule of classes or events. However, if you need to make calculations that span the daylight savings time, or if you need to deal with users in different time zones, you should use the ZonedDateTime class that we discuss next.

java.time.LocalTime8

- static LocalTime now()
  gets the current LocalTime.
- static LocalTime of(int hour, int minute)
- static LocalTime of(int hour, int minute, int second)
- static LocalTime of(int hour, int minute, int second, int

nanoOfSecond)
yield a local time with the given hour (between 0 and 23), minute, second (between 0 and 59), and nanosecond (between 0 and 999,999,999).

- LocalTime plus*Xxx* (long number)
- LocalTime minus*Xxx* (long number)
  yields a LocalTime obtained by adding or subtracting the given number of time units, where *Xxx* is Hours, Minutes, Seconds, or Nanos.
- LocalTime plus(TemporalAmount amountToAdd)
- LocalTime minus(TemporalAmount amountToSubtract)
  yield a LocalTime that is the given amount away from this LocalTime.
- LocalTime with*Xxx*(int value)
  returns a new LocalTime with the time unit changed to the given value, where *Xxx* is Nano, Second, Minute, or Hour
- int getHour()
  gets the hour (between 0 and 23).
- int getMinute()
- int getSecond()
  get the minute or second (between 0 and 59).
- int getNano()
  gets the nanoseconds (between 0 and 999,999,999).
- int toSecondOfDay()
- long toNanoOfDay()
  yield the seconds or nanoseconds since midnight.
- boolean isBefore(LocalTime other)
- boolean isAfter(LocalTime other)
  return true if this date is before or after the given date.

6.5. Zoned Time

Time zones, perhaps because they are an entirely human creation, are even messier than the complications caused by the earth's irregular rotation. In a rational world, we'd all follow the clock in Greenwich, and some of us would eat our lunch at 02:00, others at 22:00. Our stomachs would figure it

out. This is actually done in China, which spans four conventional time zones. Elsewhere, we have time zones with irregular and shifting boundaries and, to make matters worse, the daylight savings time.

As capricious as the time zones may appear to the enlightened, they are a fact of life. When you implement a calendar application, it needs to work for people who fly from one country to another. When you have a conference call at 10:00 in New York, but happen to be in Berlin, you expect to be alerted at the correct local time.

The Internet Assigned Numbers Authority (IANA) keeps a database of all known time zones around the world (https://www.iana.org/time-zones), which is updated several times per year. The bulk of the updates deals with the changing rules for daylight savings time. Java uses the IANA database.

Each time zone has an ID, such as America/New_York or Europe/Berlin. To find out all available time zones, call ZoneId.getAvailableZoneIds. At the time of this writing, there are almost 600 IDs.

Given a time zone ID, the static method ZoneId.of(id) yields a ZoneId object. You can use that object to turn a LocalDateTime object into a ZonedDateTime object by calling local.atZone(zoneId), or you can construct a ZonedDateTime by calling the static method ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId). For example,

```
ZonedDateTime apollo11launch  = ZonedDateTime.of(1969, 7, 16, 9, 3
   ZoneId.of("America/New_York"));
   // 1969-07-16T09:32-04:00[America/New_York]
```

This is a specific instant in time. Call apollo11launch.toInstant to get the Instant. Conversely, if you have an instant in time, call instant.atZone(ZoneId.of("UTC")) to get the ZonedDateTime at the Greenwich Royal Observatory, or use another ZoneId to get it elsewhere on the planet.

> UTC stands for “Coordinated Universal Time,” and the acronym is a compromise between the aforementioned English and the French “Temps Universel Coordiné,” having the distinction of being incorrect in either language. UTC is the time at the Greenwich Royal Observatory, without daylight savings time.

Many of the methods of ZonedDateTime are the same as those of LocalDateTime (see the API notes at the end of this section). Most are straightforward, but daylight savings time introduces some complications.

When daylight savings time starts, clocks advance by an hour. What happens when you construct a time that falls into the skipped hour? For example, in 2013, Central Europe switched to daylight savings time on March 31 at 2:00. If you try to construct nonexistent time March 31 2:30, you actually get 3:30.

```
ZonedDateTime skipped = ZonedDateTime.of(
   LocalDate.of(2013, 3, 31),
   LocalTime.of(2, 30),
   ZoneId.of("Europe/Berlin"));
   // Constructs March 31 3:30
```

Conversely, when daylight time ends, clocks are set back by an hour, and there are two instants with the same local time! When you construct a time within that span, you get the earlier of the two.

```
ZonedDateTime ambiguous = ZonedDateTime.of(
   LocalDate.of(2013, 10, 27), // End of daylight savings time
   LocalTime.of(2, 30),
   ZoneId.of("Europe/Berlin"));
   // 2013-10-27T02:30+02:00[Europe/Berlin]
ZonedDateTime anHourLater = ambiguous.plusHours(1);
   // 2013-10-27T02:30+01:00[Europe/Berlin]
```

An hour later, the time has the same hours and minutes, but the zone offset has changed.

You also need to pay attention when adjusting a date across

daylight savings time boundaries. For example, if you set a meeting for next week, don't add a duration of seven days:

```
ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
   // Caution! Won't work with daylight savings time
```

Instead, use the Period class.

```
ZonedDateTime nextMeeting = meeting.plus(Period.ofDays(7)); // OK
```

There is also an OffsetDateTime class that represents times with an offset from UTC, but without time zone rules. That class is intended for specialized applications that specifically require the absence of those rules, such as certain network protocols. For human time, use ZonedDateTime.

The example program in Listing 6.3 demonstrates the ZonedDateTime class.

Listing 6.3 **v2ch06/ZonedTimes.java**

```
package v2ch06;

/**
 * This program demonstrates zoned times.
 */

import module java.base;

class ZonedTimes {
   void main() {
      ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0
            ZoneId.of("America/New_York")); // 1969-07-16T09:32-04:00[America/N
      IO.println("apollo11launch: " + apollo11launch);

      Instant instant = apollo11launch.toInstant();
      IO.println("instant: " + instant);

      ZonedDateTime zonedDateTime = instant.atZone(ZoneId.of("UTC"));
```

```
        IO.println("zonedDateTime: " + zonedDateTime);

        ZonedDateTime skipped = ZonedDateTime.of(
              LocalDate.of(2013, 3, 31), LocalTime.of(2, 30),
              ZoneId.of("Europe/Berlin")); // Constructs March 31 3:30
        IO.println("skipped: " + skipped);

        ZonedDateTime ambiguous = ZonedDateTime.of(
              LocalDate.of(2013, 10, 27), // End of daylight savings time
              LocalTime.of(2, 30), ZoneId.of("Europe/Berlin"));
        // 2013-10-27T02:30+02:00[Europe/Berlin]
        ZonedDateTime anHourLater = ambiguous.plusHours(1);
        // 2013-10-27T02:30+01:00[Europe/Berlin]
        IO.println("ambiguous: " + ambiguous);
        IO.println("anHourLater: " + anHourLater);

        ZonedDateTime meeting = ZonedDateTime.of(LocalDate.of(2013, 10, 31),
              LocalTime.of(14, 30), ZoneId.of("America/Los_Angeles"));
        IO.println("meeting: " + meeting);
        ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
        // Caution! Won't work with daylight savings time
        IO.println("nextMeeting: " + nextMeeting);
        nextMeeting = meeting.plus(Period.ofDays(7)); // OK
        IO.println("nextMeeting: " + nextMeeting);
    }
}
```

java.time.ZonedDateTime[8]

- static ZonedDateTime now()
  gets the current ZonedDateTime.
- static ZonedDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond, ZoneId zone)
- static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone)
- static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)
- static ZonedDateTime ofInstant(Instant instant, ZoneId zone)
  yield a ZonedDateTime with the given parameters and time zone.
- ZonedDateTime plus*Xxx*(long number)

- ZonedDateTime minus*Xxx*(long number)
  yields a ZonedDateTime obtained by adding or subtracting the given number of time units, where *Xxx* is Nanos, Seconds, Minutes, Hours, Days, Weeks, Months, or Years.
- ZonedDateTime plus(TemporalAmount amountToAdd)
- ZonedDateTime minus(TemporalAmount amountToSubtract)
  yield a ZonedDateTime that is the given amount away from this ZonedDateTime.
- ZonedDateTime with*Xxx*(int value)
  returns a new ZonedDateTime with the given temporal unit replaced by the given value, where *Xxx* is Nano, Second, Minute, Hour, DayOfMonth, DayOfYear, Month, or Year.
- ZonedDateTime withZoneSameInstant(ZoneId zone)
- ZonedDateTime withZoneSameLocal(ZoneId zone)
  return a new ZonedDateTime in the given time zone, either representing the same instant or the same local time.
- int getDayOfMonth()
  gets the day of the month (between 1 and 31).
- int getDayOfYear()
  gets the day of the year (between 1 and 366).
- DayOfWeek getDayOfWeek()
  gets the day of the week, returning a value of the DayOfWeek enumeration.
- Month getMonth()
- int getMonthValue()
  get the month as a value of the Month enumeration, or as a number between 1 and 12.
- int getYear()
  gets the year, between –999,999,999 and 999,999,999.
- int getHour()
  gets the hour (between 0 and 23).
- int getMinute()
- int getSecond()
  get the minute or second (between 0 and 59).
- int getNano()
  gets the nanoseconds (between 0 and 999,999,999).
- public ZoneOffset getOffset()

gets the offset from UTC. Offsets can vary from –12:00 to +14:00. Some time zones have fractional offsets. Offsets change with daylight savings time.

- LocalDate toLocalDate()
- LocalTime toLocalTime()
- LocalDateTime toLocalDateTime()
- Instant toInstant()
  yield the local date, time, or date/time, or the corresponding instant.
- boolean isBefore(ChronoZonedDateTime other)
- boolean isAfter(ChronoZonedDateTime other)
  return true if this zoned date/time is before or after the given zoned date/time.

6.6. Formatting and Parsing

The DateTimeFormatter class provides three kinds of formatters to print a date/time value:

- Predefined standard formatters (see Table 6.1)
- Locale-specific formatters
- Formatters with custom patterns

To use one of the standard formatters, simply call its format method:

```
String formatted = DateTimeFormatter.ISO_OFFSET_DATE_TIME.forr
   // 1969-07-16T09:32:00-04:00"
```

Formatter	Description	Example
BASIC_ISO_DATE	Year, month, day, zone offset without separators	19690716-0400
ISO_LOCAL_DATE, ISO_LOCAL_TIME, ISO_LOCAL_DATE_TIME	Separators -, :, T	1969-07-16, 09:32:00, 1969-07-16T09:32:00
ISO_OFFSET_DATE, ISO_OFFSET_TIME	like ISO_LOCAL_XXX,	1969-07-16-04:00, 09:32:00-04:00,

ISO_OFFSET_DATE_TIME	Time with zone offset	1969-07-16T09:32:00-04:00
ISO_ZONED_DATE_TIME	Time, zone offset and zone ID	1969-07-16T09:32:00-04:00[America/New_York]
ISO_INSTANT	In UTC, denoted by the Z zone ID	1969-07-16T13:32:00Z
ISO_DATE, ISO_TIME, ISO_DATE_TIME	Like ISO_OFFSET_DATE, ISO_OFFSET_TIME, and ISO_ZONED_DATE_TIME, but the zone information is optional	1969-07-16-04:00, 09:32:00-04:00, 1969-07-16T09:32:00-04:00[America/New_York]
ISO_ORDINAL_DATE	The year and day of year, for LocalDate	1969-197-04:00
ISO_WEEK_DATE	The year, week, and day of week, for LocalDate	1969-W29-3-04:00
RFC_1123_DATE_TIME	The standard for email timestamps, codified in RFC 822 and updated to four digits for the year in RFC 1123	Wed, 16 Jul 1969 09:32:00 -0400

Table 6.1: Predefined Formatters

The standard formatters are mostly intended for machine-readable timestamps. To present dates and times to human readers, use a locale-specific formatter. There are four styles, SHORT, MEDIUM, LONG, and FULL, for both date and time—see Table 6.2.

The static methods ofLocalizedDate, ofLocalizedTime, and ofLocalizedDateTime create such a formatter. For example:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate
```

```
String formatted = formatter.format(apollo11launch);
   // July 16, 1969 9:32:00 AM EDT
```

These methods use the default locale. To change to a different locale, simply use the withLocale method.

```
formatted = formatter.withLocale(Locale.FRENCH).format(apollo11la
   // 16 juillet 1969 09:32:00 EDT
```

The DayOfWeek and Month enumerations have methods getDisplayName for giving the names of weekdays and months in different locales and formats.

```
for (DayOfWeek w : DayOfWeek.values())
   IO.print(w.getDisplayName(TextStyle.SHORT, Locale.ENGLISH) + "
   // Prints Mon Tue Wed Thu Fri Sat Sun
```

See Chapter 7 for more information about locales.

Style	Date	Time
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT

Table 6.2: Locale-Specific Formatting Styles

The java.time.format.DateTimeFormatter class is intended as a replacement for java.util.DateFormat. If you need an instance of the latter for backward compatibility, call formatter.toFormat().

Finally, you can roll your own date format by specifying a pattern. For example,

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

formats a date in the form Wed 1969-07-16 09:32. Each letter denotes a different time field, and the number of times the

letter is repeated selects a particular format, according to rules that are arcane and seem to have organically grown over time. Table 6.3 shows the most useful pattern elements.

ChronoField or Purpose	Examples
ERA	G: AD, GGGG: Anno Domini, GGGGG: A
YEAR_OF_ERA	yy: 69, yyyy: 1969
MONTH_OF_YEAR	M: 7, MM: 07, MMM: Jul, MMMM: July, MMMMM: J
DAY_OF_MONTH	d: 6, dd: 06
DAY_OF_WEEK	e: 3, E: Wed, EEEE: Wednesday, EEEEE: W
HOUR_OF_DAY	H: 9, HH: 09
CLOCK_HOUR_OF_AM_PM	h: 9, hh: 09
AMPM_OF_DAY	a: AM
Period of day (since Java 16)	B: in the evening, at night, midnight
MINUTE_OF_HOUR	mm: 02
SECOND_OF_MINUTE	ss: 00
NANO_OF_SECOND	nnnnnn: 000000
Time zone ID	VV: America/New_York
Time zone name	z: EDT, zzzz: Eastern Daylight Time, v: ET, vvvv: Eastern Time
Zone offset	x: -04, xx: -0400, xxx: -04:00, XXX: same, but use Z for zero

Table 6.3: Commonly Used Formatting Symbols for Date/Time Formats

---

Pay attention to the capitalization of these letters: M is months, m is minutes. You probably want the day of the month d, and not the day of the year D, and seconds s rather than fractions of a second S. Most perilously, y indicates a normal year but Y is a “week-numbering year,” which is subtly different if January 1 falls on a Friday, Saturday, or Sunday.

The ofLocalizedPattern method yields a formatter that assembles parts of patterns in a locale-specific way, with the appropriate separators and ordering of the parts. For example:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedPatt
formatter.withLocale(Locale.US).format(apollo11launch) // Jul 16, 19
formatter.withLocale(Locale.GERMANY).format(apollo11launch) // 1
formatter.withLocale(Locale.CHINA).format(apollo11launch) // 1969
```

You only specify the symbols for the parts, in descending size. Do not include any separators. For the hours, use j for a localized 12- or 24-hour format:

```
formatter = DateTimeFormatter.ofLocalizedPattern("jm");
formatter.withLocale(Locale.US).format(LocalTime.of(23, 59)) // 11:5
formatter.withLocale(Locale.GERMANY).format(LocalTime.of(23, 59)
```

To parse a date/time value from a string, use one of the static parse methods. For example,

```
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
ZonedDateTime apollo11launch =
   ZonedDateTime.parse("1969-07-16 03:32:00-0400",
      DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
```

The first call uses the standard ISO_LOCAL_DATE formatter, the second one a custom formatter.

The program in Listing 6.4 shows how to format and parse dates and times.

Listing 6.4 **v2ch06/Formatting.java**

```
package v2ch06;

/**
 * This program demonstrates time formatting.
 */

import module java.base;
```

```
class Formatting {
   void main() {
      ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0
            ZoneId.of("America/New_York"));

      String formatted = DateTimeFormatter.ISO_OFFSET_DATE_TIME.format(ap
      // 1969-07-16T09:32:00-04:00
      IO.println(formatted);

      DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDateTime
      formatted = formatter.format(apollo11launch);
      // July 16, 1969 9:32:00 AM EDT
      IO.println(formatted);
      formatted = formatter.withLocale(Locale.FRENCH).format(apollo11launch
      // 16 juillet 1969 09:32:00 EDT
      IO.println(formatted);

      formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
      formatted = formatter.format(apollo11launch);
      IO.println(formatted);

      LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
      IO.println("churchsBirthday: " + churchsBirthday);
      apollo11launch = ZonedDateTime.parse("1969-07-16 03:32:00-0400",
            DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
      IO.println("apollo11launch: " + apollo11launch);

      for (DayOfWeek w : DayOfWeek.values())
         IO.print(w.getDisplayName(TextStyle.SHORT, Locale.ENGLISH) + " ");
   }
}
```

java.time.format.DateTimeFormatter8

- String format(TemporalAccessor temporal)
  formats the given value. The classes Instant, LocalDate, LocalTime, LocalDateTime, and ZonedDateTime, as well as many others, implement the TemporalAccessor interface.
- static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)
- static DateTimeFormatter ofLocalizedTime(FormatStyle timeStyle)

- static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)
- static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle timeStyle)
  yield a formatter for the given styles. The FormatStyle enumeration has values SHORT, MEDIUM, LONG, and FULL.
- static DateTimeFormatter ofLocalizedPattern(String requestedTemplate)19
  creates a formatter that can be adapted to a locale, using the parts that are specified in the template.
- DateTimeFormatter withLocale(Locale locale)
  yields a formatter equal to this formatter, using the given locale.
- static DateTimeFormatter ofPattern(String pattern)
- static DateTimeFormatter ofPattern(String pattern, Locale locale)
  yield a formatter using the given pattern and locale. See for the pattern syntax.

java.time.LocalDate8

- static LocalDate parse(CharSequence text)
- static LocalDate parse(CharSequence text, DateTimeFormatter formatter)
  yield a LocalDate using the default formatter, or the given formatter.

java.time.ZonedDateTime8

- static ZonedDateTime parse(CharSequence text)
- static ZonedDateTime parse(CharSequence text, DateTimeFormatter formatter)
  yield a ZonedDateTime using the default formatter, or the given formatter.

6.7. Interoperating with Legacy Code

As a new creation, the Java Date and Time API will have to interoperate with existing classes—in particular, the ubiquitous java.util.Date, java.util.GregorianCalendar, and java.sql.Date/Time/Timestamp.

The Instant class is a close analog to java.util.Date. The toInstant method of the legacy Date class yields an Instant, and the static from method converts in the other direction.

Similarly, ZonedDateTime is a close analog to java.util.GregorianCalendar, and that legacy class also has conversion methods. The toZonedDateTime method converts a GregorianCalendar to a ZonedDateTime, and the static from method does the opposite conversion.

Another set of conversions is available for the date and time classes in the java.sql package. You can also pass a DateTimeFormatter to legacy code that uses java.text.Format. Table 6.4 summarizes these conversions.

Classes	To Legacy Class	From Legacy Class
Instant ↔ java.util.Date	Date.from(instant)	date.toInstant()
ZonedDateTime ↔ java.util.GregorianCalendar	GregorianCalendar.from(zonedDateTime)	cal.toZonedDateTime()
Instant ↔ java.sql.Timestamp	Timestamp.from(instant)	stamp.toInstant()
LocalDateTime ↔ java.sql.Timestamp	Timestamp.valueOf(localDateTime)	stamp.toLocalDateTime()
LocalDate ↔ java.sql.Date	Date.valueOf(localDate)	date.toLocalDate()
LocalTime ↔ java.sql.Time	Time.valueOf(localTime)	time.toLocalTime()
DateTimeFormatter	formatter.toFormat()	None

→ java.text.DateFormat		
java.util.TimeZone ↔ ZoneId	TimeZone.getTimeZone(id)	zone.toZoneId()
java.nio.file.attribute.FileTime ↔ Instant	FileTime.from(instant)	fileTime.toInstant()

Table 6.4: Conversions between java.time Classes and Legacy Classes

You now know how to use the date and time library to work with date and time values around the world. The next chapter takes the discussion of programming for an international audience further. You will see how to format program messages, numbers, and currencies in the way that makes sense for your customers, wherever they may be.

Chapter 7 ▪ Internationalizati

There’s a big world out there; we hope that lots of its inhabitants will be interested in your software. The Internet, after all, effortlessly spans the barriers between countries. On the other hand, when you pay no attention to an international audience, you are putting up a barrier.

The Java programming language was the first language designed from the ground up to support internationalization. From the beginning, it had the one essential feature needed for effective internationalization: It used Unicode for all strings. Unicode support makes it easy to write Java programs that manipulate strings in most of the world's languages.

Many programmers believe that all they need to do to internationalize their application is to support Unicode and to translate the messages in the user interface. However, as this chapter demonstrates, there is a lot more to internationalizing programs than just Unicode support. Dates, times, currencies, even numbers are formatted differently in different parts of the world. You need an easy way to configure menu and button names, message strings, and keyboard shortcuts for different languages.

In this chapter, you will see how to write internationalized Java programs and how to localize dates, times, numbers, text, and GUIs. I will show you the tools that Java offers for writing internationalized programs. The chapter concludes with a complete example—a retirement calculator with a user

interface in English, German, and Chinese.

7.1. Locales

When you look at an application that is adapted to an international market, the most obvious difference you notice is the language. This observation is actually a bit too limiting for true internationalization, since countries can share a common language, but you might still need to do some work to make computer users of both countries happy. As Oscar Wilde famously said: “We have really everything in common with America nowadays, except, of course, language.”

7.1.1. Why Locales?

When you provide international versions of a program, all program messages need to be translated to the local language. However, simply translating the user interface text is not sufficient. There are many more subtle differences—for example, numbers are formatted quite differently in English and German. The number

123,456.78

should be displayed as

123.456,78

for a German user—that is, the roles of the decimal point and the decimal comma separator are reversed. There are similar variations in the display of dates. In the United States, dates are (somewhat irrationally) displayed as month/day/year. Germany uses the more sensible order of day/month/year, whereas in China, the usage is year/month/day. Thus, the date

3/22/61

should be presented as

22.03.1961

to a German user. Of course, if the month names are written out explicitly, the difference in languages becomes apparent. The English

> March 22, 1961

should be presented as

> 22. März 1961

in German, or

> 1961 年 3 月 22 日

in Chinese.

A locale captures local preferences such as these. Whenever you present numbers, dates, currency values, and other items whose formatting varies by language or location, you need to use locale-aware APIs.

7.1.2. Specifying Locales

A locale is made up of up to five components:

1. A language, specified by two or three lowercase letters, such as en (English), de (German), or zh (Chinese). Table 7.1 shows common codes.
2. Optionally, a script, specified by four letters with an initial uppercase, such as Latn (Latin), Cyrl (Cyrillic), or Hant (traditional Chinese characters). This can be useful because some languages, such as Serbian, are written in Latin or Cyrillic, and some Chinese readers prefer the traditional over the simplified characters.
3. Optionally, a country or region, specified by two uppercase letters or three digits, such as US (United States) or CH (Switzerland). Table 7.2 shows common codes.
4. Optionally, a variant, specifying miscellaneous features such as dialects or spelling rules. Variants are rarely used nowadays. There used to be a “Nynorsk” variant of

Norwegian, but it is now expressed with a different language code, nn. What used to be variants for the Japanese imperial calendar and Thai numerals are now expressed as extensions (see the next item).

5. Optionally, an extension. Extensions describe local preferences for calendars (such as the Japanese calendar), numbers (Thai instead of Western digits), and so on. The Unicode standard specifies some of these extensions. Extensions start with u- and a two-letter code specifying whether the extension deals with the calendar (ca), numbers (nu), and so on. For example, the extension u-nu-thai denotes the use of Thai numerals. Other extensions are entirely arbitrary and start with x-, such as x-java.

Rules for locales are formulated in the “Best Current Practices” memo BCP 47 of the Internet Engineering Task Force (https://www.rfc-editor.org/info/bcp47). You can find a more accessible summary at https://www.w3.org/International/articles/language-tags.

Code	Language

Table 7.1: Common ISO 639-1 Language Codes

Code	Country

For your convenience, there are predefined locale objects for various countries:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

A number of predefined locales specify just a language without a location:

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

The Locale.ROOT has language-neutral rules that can be useful for machine-processed strings.

Finally, the static getAvailableLocales method returns an array of all locales known to the virtual machine. The availableLocales method returns a stream instead.

> You can get all language codes as Locale.getISOLanguages() and all country codes as Locale.getISOCountries().

7.1.3. The Default Locale

The static getDefault method of the Locale class initially gets the default locale as stored by the local operating system. You can change the default Java locale by calling the setDefault method with a different locale.

Some operating systems allow the user to specify different locales for displayed messages and for formatting. For example, a French speaker living in the United States can have French menus but currency values in dollar.

To obtain these preferences, call

```
Locale displayLocale = Locale.getDefault(Locale.Category.DISPLAY);
Locale formatLocale = Locale.getDefault(Locale.Category.FORMAT);
```

In UNIX, you can specify separate locales for numbers, currencies, and dates, by setting the LC_NUMERIC, LC_MONETARY, and LC_TIME environment variables. Java does not pay attention to these settings.

If you want to test a locale that just has language and country settings, you can supply them on the command line when you launch your program. For example, here we set the default locale to de-CH:

```
java -Duser.language=de -Duser.region=CH MyProgram
```

7.1.4. Display Names

Once you have a locale, what can you do with it? Initially, not much, as it turns out. The only useful methods in the Locale class are those for identifying the language and country codes. The most important one is getDisplayName. It returns a string describing the locale. This string does not contain the cryptic two-letter codes, but is in a form that can be presented to a user, such as

```
German (Switzerland)
```

Actually, there is a problem here. The display name is issued in the default locale. That might not be appropriate. If your user already selected German as the preferred language, you probably want to present the string in German. You can do just that by passing the German locale as an argument. The code

```
Locale locale = Locale.of("de", "CH");
IO.println(locale.getDisplayName(Locale.GERMAN));
```

prints

```
Deutsch (Schweiz)
```

This example shows why you need Locale objects. You feed them to locale-aware methods that produce text that is presented to users in different locations. You will see many examples of this in the following sections.

> Even such mundane operations as turning a string into lowercase or uppercase can be locale-specific. For example, in the Turkish locale, the lowercase of the letter I is a dotless ı. Programs that tried to normalize strings by storing them in lowercase have mysteriously failed for Turkish customers where I and the dotted i don't have the same lowercase version. It is a good idea to always use the variants of toUpperCase and toLowerCase with a Locale parameter. For example, try out:
>
> ```
> String command = "QUIT".toLowerCase(Locale.of("tr"));
>    // "quıt" with a dotless ı
> ```
>
> Of course, in Turkey, where Locale.getDefault() yields just that locale, "QUIT".toLowerCase() is not the same as "quit".
>
> If you want to normalize English language strings to lowercase, you should pass an English locale to the toLowerCase method. For language-neutral processing,

---

use Locale.ROOT.

You can explicitly set the locale for input and output operations.

- When reading numbers from a Scanner, you can set its locale with the useLocale method.
- The String.format and PrintWriter.printf methods have overloaded forms with a Locale parameter.

java.util.Locale1.1

- static Locale.of(String language)19
- static Locale.of(String language, String country)19
  construct a locale with the given language and country.
- static Locale forLanguageTag(String languageTag)7
  constructs a locale corresponding to the given language tag.
- static Locale getDefault()
  returns the default locale.
- static void setDefault(Locale newLocale)
  sets the default locale.
- String getDisplayName()
  returns a name describing the locale, expressed in the current locale.
- String getDisplayName(Locale inLocale)
  returns a name describing the locale, expressed in the given locale.
- String getLanguage()
  returns the language code—a lowercase two-letter ISO 639 code.
- String getDisplayLanguage()
  returns the name of the language, expressed in the current locale.
- String getDisplayLanguage(Locale inLocale)
  returns the name of the language, expressed in the given locale.
- String getCountry()

returns the country code as an uppercase two-letter ISO 3166 code.

- static String[] getISOCountries()
- static Set<String> getISOCountries(Locale.IsoCountryCode type)9
  get all two-letter country codes, or all country codes with 2, 3, or 4 letters. The type is one of the enumeration constants PART1_ALPHA2, PART1_ALPHA3, and PART3.
- String getDisplayCountry()
  returns the name of the country, expressed in the current locale.
- String getDisplayCountry(Locale inLocale)
  returns the name of the country, expressed in the given locale.
- String toLanguageTag()7
  returns the IETF BCP 47 language tag for this locale—for example, "de-CH".
- String toString()
  returns a description of the locale, with the language and country separated by underscores (e.g., "de_CH"). Use this method only for debugging.

7.2. Number Formats

I already mentioned how number and currency formatting is highly locale-dependent. The Java library supplies a collection of formatter objects that can format and parse numeric values in the java.text package.

7.2.1. Formatting Numeric Values

Go through the following steps to format a number for a particular locale:

1. Get the locale object, as described in the preceding section.
2. Use a “factory method” to obtain a formatter object.
3. Use the formatter object for formatting and parsing.

The factory methods are static methods of the NumberFormat class with a Locale parameter. There are three factory methods: getNumberInstance, getCurrencyInstance, and getPercentInstance. These methods return objects that can format and parse numbers, currency amounts, and percentages, respectively. For example, here is how you can format a currency value in German:

```
Locale locale = Locale.GERMAN;
NumberFormat formatter = NumberFormat.getCurrencyInstance(loc
double amount = 123456.78;
String result = formatter.format(amount);
```

The result is

```
123.456,78 €
```

Note that the currency symbol is € and that it is placed at the end of the string. Also, note the reversal of decimal points and decimal commas.

Java 12 adds a “compact” format with two styles: short (123K) and long (123 thousand). Here is how to obtain the short style:

```
formatter = NumberFormat.getCompactNumberInstance(locale, Nur
```

Conversely, to read in a number that was entered or stored with the conventions of a certain locale, use the parse method. For example, the following code parses the value that the user typed into a text field. The parse method can deal with decimal points and commas, as well as digits in other languages.

```
TextField inputField;
. . .
NumberFormat formatter = NumberFormat.getNumberInstance();
// get the number formatter for default locale
Number input = formatter.parse(inputField.getText().strip());
double x = input.doubleValue();
```

The return type of parse is the abstract type Number. The returned object is either a Double or a Long wrapper object,

depending on whether the parsed number was a floating-point number. If you don't care about the distinction, you can simply use the doubleValue method of the Number class to retrieve the wrapped number.

Objects of type Number are not automatically unboxed—you cannot simply assign a Number object to a primitive type. Instead, use the doubleValue or intValue method.

If the text for the number is not in the correct form, the method throws a ParseException. For example, leading whitespace in the string is not allowed. (Call strip to remove it.) However, any characters that follow the number in the string are simply ignored, so no exception is thrown.

Note that the instances returned by the get*Xxx*Instance factory methods are not actually of type NumberFormat. The NumberFormat type is an abstract class, and the actual formatters belong to one of its subclasses. The factory methods merely know how to locate the object that belongs to a particular locale.

You can get a list of the currently supported locales with the static getAvailableLocales method. That method returns an array of the locales for which number formatter objects can be obtained.

You can use a Scanner to read localized integers and floating-point numbers. Call the useLocale method to set the locale.

The book companion code contains a GUI program that lets you experiment with number formatters (see Figure 7.1). The combo box at the top of the figure contains all locales with number formatters. You can choose between number, currency, and percentage formatters. Each time you make another choice, the number in the text field is reformatted. If you go through a few locales, you can get a good impression of the many ways that a number or currency value can be formatted.

You can also type a different number and click the Parse button to call the parse method, which tries to parse what you entered. If your input is successfully parsed, it is passed to format and the result is displayed. If parsing fails, a “Parse error” message is displayed in the text field.



Figure 7.1: The NumberFormatDemo program

Listing 7.1 shows a text version of the number format explorer that is a bit easier to follow.

Listing 7.1 **v2ch07/numberFormat/NumberFormatDemo2.java**

```
package v2ch07.numberFormat;

import module java.base;
import v2ch07.util.Choices;

/**
 * This program demonstrates formatting numbers under various locales.
 */
class NumberFormatDemo2 {
   void main() {
      var locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
      Arrays.sort(locales, Comparator.comparing(Locale::getDisplayName));
      Locale locale = Choices.choose(locales, Locale::getDisplayName);

      var formatters = new LinkedHashMap<NumberFormat, String>();
      formatters.put(NumberFormat.getNumberInstance(locale), "Number");
      formatters.put(NumberFormat.getCompactNumberInstance(locale, Numb
            "Compact Short");
      formatters.put(NumberFormat.getCompactNumberInstance(locale, Numb
            "Compact Long");
```

```
      formatters.put(NumberFormat.getPercentInstance(locale), "Percent");
      formatters.put(NumberFormat.getCurrencyInstance(locale), "Currency");
      NumberFormat formatter = Choices.choose(formatters);

      String operation = Choices.choose("Format", "Parse");
      if (operation.equals("Format")) {
         double number = Double.parseDouble(
               IO.readln("Enter a floating-point number to format: "));
         System.out.println(formatter.format(number));
      }
      else {
         String text = IO.readln("Enter a floating-point number to parse: ");
         try {
            System.out.println(formatter.parse(text));
         }
         catch (ParseException e) {
            System.out.println("ParseException " + e.getMessage());
         }
      }
   }
}
```

java.text.NumberFormat1.1

- static Locale[] getAvailableLocales()
  returns an array of Locale objects for which NumberFormat formatters are available.
- static NumberFormat getNumberInstance()
- static NumberFormat getNumberInstance(Locale inLocale)
- static NumberFormat getCurrencyInstance()
- static NumberFormat getCurrencyInstance(Locale inLocale)
- static NumberFormat getPercentInstance()
- static NumberFormat getPercentInstance(Locale inLocale)
  return a formatter for numbers, currency amounts, or percentage values for the current locale or for the given locale.
- static NumberFormat getCompactNumberInstance()12
- static NumberFormat getCompactNumberInstance(Locale locale, NumberFormat.Style formatStyle)12
  return a compact number formatter in the current locale

and SHORT style, or in the given locale and style (either SHORT or LONG).

- String format(double number)
- String format(long number)
  return the string resulting from formatting the given floating-point number or integer.
- Number parse(String source)
  parses the given string and returns the number value, as a Double if the input string describes a floating-point number or as a Long otherwise. The beginning of the string must contain a number; no leading whitespace is allowed. The number can be followed by other characters, which are ignored. Throws ParseException if parsing was not successful.
- void setParseIntegerOnly(boolean value)
- boolean isParseIntegerOnly()
  set or get a flag to indicate whether this formatter should parse only integer values.
- void setGroupingUsed(boolean newValue)
- boolean isGroupingUsed()
  set or get a flag to indicate whether this formatter emits and recognizes decimal separators (such as 100,000).
- void setMinimumIntegerDigits(int newValue)
- int getMinimumIntegerDigits()
- void setMaximumIntegerDigits(int newValue)
- int getMaximumIntegerDigits()
- void setMinimumFractionDigits(int newValue)
- int getMinimumFractionDigits()
- void setMaximumFractionDigits(int newValue)
- int getMaximumFractionDigits()
  set or get the maximum or minimum number of digits allowed in the integer or fractional part of a number.

7.2.2. The DecimalFormat Class

In most locales, the NumberFormat factory methods that you saw in the preceding section return an instance of the DecimalFormat class. This class describes the formatting

variations that occur around the world. You can modify individual settings, or create an entirely new formatter. A pattern syntax makes this a bit more convenient.

The pattern describes the number of required and optional digits, and the prefixes and suffixes for positive and negative numbers. There are also a couple more esoteric settings—see Table 7.3.

Consider this example:

```
var formatter = new DecimalFormat("0.00;(#)");
```

The semicolon separates the positive and the optional negative part. From the positive part, we see that there must be at least one digit in the integer part and at least two digits in the fractional part. This holds for all numbers, regardless of sign. Negative numbers use the accounting format—a prefix (and a suffix).

The prefix or suffix may contain a ¤ currency sign (U+00A4) to denote where the currency symbol should go.

If you need to insert a special character into the prefix or suffix, precede it with a single quote. For example, a prefix '# is a literal hash sign, and o''clock has one quotation mark.

Property Name	Description	Pattern
groupingSize	The number of integer digits grouped together (usually 3 or 4)	The number of digits between the last , and the end of the integer part, such as #,### for a grouping size of 3.
minimum/ maximumFractionDigits	The minimum and maximum number of digits of the fractional part	Use required (0) and optional (#) digits in the fractional part: .00##.
minimumIntegerDigits	The minimum	Use required (0)

maximumIntegerDigits	The maximum number of integer digits	digits in the integer part: 000. With exponential notation, the number of # digits in the integer part, such as ###,##0.00E0. Otherwise, cannot be set in pattern.
—	the minimum number of digits in the exponential part	Use 0 digits in the E part: #.00E00.
multiplier	The multiplier for percent, per thousand	Use % for 100, ‰ (U+2030) for 1000.
positivePrefix/ Suffix, negativePrefix/ Suffix	The prefix and suffix for positive or negative numbers	Surround the positive or negative part of the pattern with the literal prefix or suffix, e.g. +#0.00;(#) uses a + prefix and no suffix for positive numbers and encloses negative numbers in parentheses.
decimalSeparatorAlwaysShown	Always if the separator is shown for a zero fractional part	Cannot be set in pattern.

Table 7.3: DecimalFormat Properties

The actual symbols for the digits, separators, and miscellaneous other number parts are taken from a DecimalFormatSymbols object that you can also customize. Table 7.4 lists its properties.

Property	Type	Description
currencySymbol	String	A string such as "$" or "EUR" to use where the prefix or suffix has a ¤ currency sign
decimalSeparator, monetaryDecimalSeparator	char	The decimal separator to use with numbers or currency values
exponentSeparator	String	The string before the exponent part, usually "E"
groupingSeparator, monetaryGroupingSeparator (since Java 15)	char	The group separator to use with numbers or currency values
infinity, naN	String	The strings for formatting Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY, and Double.NaN
internationalCurrencySymbol	String	The ISO 4217 currency symbol — see Section 7.2.3
minusSign	char	The minus sign that is used when there is no negative pattern
percent, perMill	char	The characters used for percent or per mille
zeroDigit	char	The character for the digit zero. The other digits are the subsequent nine Unicode characters.

Table 7.4: DecimalFormatSymbol Properties

Suppose, for example, that you want to show numbers in the US style, with , as group separator and . as decimal separator, no matter what the locale is. Customize the DecimalFormatSymbols and the formatter:

```
DecimalFormatSymbols symbols = new DecimalFormatSymbols(loca
symbols.setGroupingSeparator(',');
symbols.setDecimalSeparator('.');
DecimalFormat formatter = (DecimalFormat) NumberFormat.getNu
formatter.setDecimalFormatSymbols(symbols);
```

7.2.3. Currencies

To format a currency value, you can use the NumberFormat.getCurrencyInstance method. However, that method is not very flexible—it returns a formatter for a single currency. Suppose you prepare an invoice for an American customer in which some amounts are in dollars and others are in euros. You can't just use two formatters:

```
NumberFormat dollarFormatter = NumberFormat.getCurrencyInsta
NumberFormat euroFormatter = NumberFormat.getCurrencyInstan
```

Your invoice would look very strange, with some values shown as $100,000 and others as 100.000 €. (Note that the euro value uses a decimal point, not a comma.)

Instead, use the Currency class to control the currency used by the formatters. You can get a Currency object by passing a currency identifier to the static Currency.getInstance method. Then call the setCurrency method for each formatter. Here is how you would set up the euro formatter for your American customer:

```
NumberFormat euroFormatter = NumberFormat.getCurrencyInstan
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

The currency identifiers are defined by ISO 4217 (see https://www.iso.org/iso-4217-currency-codes.html). Table 7.5 provides

a partial list.

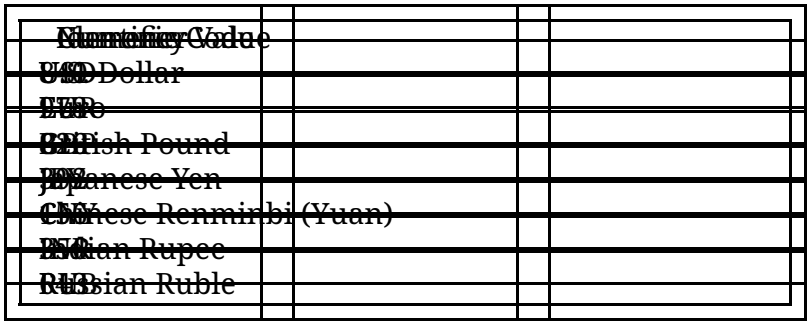


Table 7.5: Currency Identifiers

java.util.Currency 1.4

- static Currency getInstance(String currencyCode)
- static Currency getInstance(Locale locale)
  return the Currency instance for the given ISO 4217 currency code or the country of the given locale.
- String toString()
- String getCurrencyCode()
- String getNumericCode() 7
- String getNumericCodeAsString() 9
  get the ISO 4217 alphabetic or numeric currency code of this currency.
- String getSymbol()
- String getSymbol(Locale locale)
  get the formatting symbol of this currency for the default locale or the given locale. For example, the symbol for USD can be "$" or "US$", depending on the locale.
- int getDefaultFractionDigits()
  gets the default number of fraction digits of this currency.
- static Set<Currency> getAvailableCurrencies() 7
  gets all available currencies.

7.3. Date and Time

When you are formatting date and time, you should be

concerned with four locale-dependent issues:

- The names of months and weekdays should be presented in the local language.
- There will be local preferences for the order of year, month, and day.
- The Gregorian calendar might not be the local preference for expressing dates.
- The time zone of the location must be taken into account.

The DateTimeFormatter class from the java.time package handles these issues. Pick one of the formatting styles shown in Table 7.6. Then, get a formatter:

```
FormatStyle style = . . .; // One of FormatStyle.SHORT, FormatStyle.MEDIUM, FormatStyle.LONG, FormatStyle.FULL
DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocalizedDate(style);
DateTimeFormatter timeFormatter = DateTimeFormatter.ofLocalizedTime(style);
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofLocalizedDateTime(dateStyle, timeStyle);
   // or DateTimeFormatter.ofLocalizedDateTime(dateStyle, timeStyle)
```

These formatters use the current locale. To use a different locale, use the withLocale method:

```
DateTimeFormatter dateFormatter =
   DateTimeFormatter.ofLocalizedDate(style).withLocale(locale);
```

Now you can format a LocalDate, LocalDateTime, LocalTime, or ZonedDateTime:

```
ZonedDateTime appointment = . . .;
String formatted = formatter.format(appointment);
```

Style	Date	Time
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT in en-US, 9:32:00 MSZ in de-DE (only for ZonedDateTime)
FULL	Wednesday, July	9:32:00 AM EDT in

16, 1969	en-US, 9:32 Uhr MSZ in de-DE (only for ZonedDateTime)

Table 7.6: Date and Time Formatting Styles

Here we use the DateTimeFormatter class from the java.time package. There is also a legacy java.text.DateFormatter class that works with Date and Calendar objects.

You can use one of the static parse methods of LocalDate, LocalDateTime, LocalTime, or ZonedDateTime to parse a date or time in a string:

```
LocalTime time = LocalTime.parse("9:32 AM", formatter);
```

These methods are not suitable for parsing human input, at least not without preprocessing. For example, the short time formatter for the United States will parse "9:32 AM" but not "9:32AM" or "9:32 am".

Date formatters parse nonexistent dates, such as November 31, and adjust them to the last date in the given month.

Sometimes, you need to display just the names of weekdays and months, for example, in a calendar application. Call the getDisplayName method of the DayOfWeek and Month enumerations.

```
for (Month m : Month.values())
   IO.println(m.getDisplayName(textStyle, locale));
```

Table 7.7 shows the text styles. The STANDALONE versions are for display outside a formatted date. For example, in Finnish, January is “tammikuuta” inside a date, but “tammikuu” standalone.

Style	Example
FULL / FULL_STANDALONE	January
SHORT / SHORT_STANDALONE	Jan
NARROW / NARROW_STANDALONE	J

Table 7.7: Values of the java.time.format.TextStyle Enumeration

The first day of the week can be Saturday, Sunday, or Monday, depending on the locale. You can obtain it like this:

```
DayOfWeek first = WeekFields.of(locale).getFirstDayOfWeek();
```

Listing 7.2 shows the DateFormat class in action. You can select a locale and see how the date and time are formatted in different places around the world.

Figure 7.2 shows a GUI version of the program, which is included in the book companion code. You may need to install Chinese fonts to see the Chinese characters displayed.

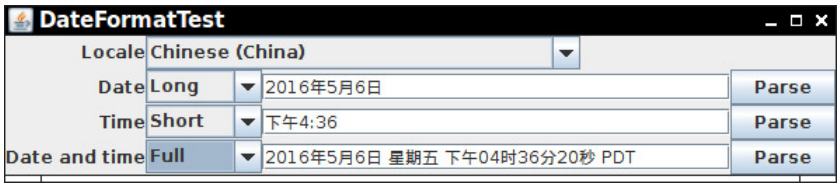


Figure 7.2: The DateFormatDemo program

Listing 7.2 **v2ch07/dateFormat/DateTimeFormatDemo2.java**

```
package v2ch07.dateFormat;

import module java.base;
import v2ch07.util.*;
```

```
/**
 * This program demonstrates formatting dates under various locales.
 */
class DateTimeFormatDemo2 {
   void main() {
      var locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
      Arrays.sort(locales, Comparator.comparing(Locale::getDisplayName));
      Locale locale = Choices.choose(locales, Locale::getDisplayName);
      FormatStyle style = Choices.choose(FormatStyle.class,
            "Short", "Medium", "Long", "Full");
      String type = Choices.choose("Date", "Time", "Date and Time");

      if (type.equals("Date")) {
         DateTimeFormatter formatter
            = DateTimeFormatter.ofLocalizedDate(style).withLocale(locale);
         System.out.println(formatter.format(LocalDate.now()));
         String input = IO.readln("Enter another date: ");
         LocalDate date = LocalDate.parse(input, formatter);
         System.out.println(formatter.format(date));
      } else if (type.equals("Time")) {
         DateTimeFormatter formatter
            = DateTimeFormatter.ofLocalizedTime(style).withLocale(locale);
         System.out.println(formatter.format(LocalTime.now()));
         String input = IO.readln("Enter another time: ");
         LocalTime time = LocalTime.parse(input, formatter);
         System.out.println(formatter.format(time));
      } else {
         DateTimeFormatter formatter
            = DateTimeFormatter.ofLocalizedDateTime(style).withLocale(locale);
         System.out.println(formatter.format(ZonedDateTime.now()));
         String input = IO.readln("Enter another date and time: ");
         ZonedDateTime dateTime = ZonedDateTime.parse(input, formatter);
         System.out.println(formatter.format(dateTime));
      }
   }
}
```

java.time.format.DateTimeFormatter 8

- static DateTimeFormatter ofLocalizedDate(FormatStyle dateStyle)
- static DateTimeFormatter ofLocalizedTime(FormatStyle dateStyle)

- static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)
- static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle timeStyle)
  return DateTimeFormatter instances that format dates, times, or dates and times with the specified styles.
- DateTimeFormatter withLocale(Locale locale)
  returns a copy of this formatter with the given locale.
- String format(TemporalAccessor temporal)
  returns the string resulting from formatting the given date/time.

java.time.LocalDate8

java.time.LocalTime8

java.time.LocalDateTime8

java.time.ZonedDateTime8

- static *Xxx* parse(CharSequence text, DateTimeFormatter formatter)
  parses the given string and returns the LocalDate, LocalTime, LocalDateTime, or ZonedDateTime described in it. Throws a DateTimeParseException if parsing was not successful.

7.4. Collation and Normalization

Most programmers know how to compare strings with the compareTo method of the String class. Unfortunately, when interacting with human users, this method is not very useful. The compareTo method uses the values of the UTF-16 encoding of the string, which leads to absurd results, even in English. For example, the following five strings are ordered according to the

compareTo method:

```
America
Zulu
able
zebra
Ångström
```

For dictionary ordering, you would want to consider upper case and lower case to be equivalent. To an English speaker, the sample list of words would be ordered as

```
able
America
Ångström
zebra
Zulu
```

However, that order would not be acceptable to a Swedish user. In Swedish, the letter Å is different from the letter A, and it is collated after the letter Z! That is, a Swedish user would want the words to be sorted as

```
able
America
zebra
Zulu
Ångström
```

To obtain a locale-sensitive comparator, call the static Collator.getInstance method:

```
Collator coll = Collator.getInstance(locale);
words.sort(coll); // Collator implements Comparator<Object>
```

Since Java 21, you can specify unusual collators with an extension. For example, the locale with language tag sv-u-co-trad uses traditional Swedish collation, which does not distinguish between “v” and “w”.

Since the Collator class implements the Comparator interface, you can pass a Collator object to the List.sort(Comparator) method to sort a list of strings.

There are a couple of advanced settings for collators. You can set a collator's strength to adjust how selective it should be. Character differences are classified as primary, secondary, or tertiary. For example, in English, the difference between "A" and "Z" is considered primary, the difference between "A" and "Å" is secondary, and between "A" and "a" is tertiary.

By setting the strength of the collator to Collator.PRIMARY, you tell it to pay attention only to primary differences. By setting the strength to Collator.SECONDARY, you instruct the collator to take secondary differences into account. That is, two strings will be more likely to be considered different when the strength is set to "secondary" or "tertiary," as shown in Table 7.8.

Primary	Secondary	Tertiary
A ≠ Z	A ≠ Å	A ≠ a
Angstrom ≠ Ångström	Angstrom = Ångström	Angstrom ≠ Ångström
Able ≠ able	Able ≠ able	Able ≠ able

Table 7.8: Collations with Different Strengths (English Locale)

When the strength has been set to Collator.IDENTICAL, no differences are allowed. This setting is mainly useful in conjunction with a rather technical collator setting, the decomposition mode, which we take up next.

Occasionally, a character or sequence of characters can be described in more than one way in Unicode. For example, an "Å" can be Unicode character U+00C5, or it can be expressed as a plain "A" (U+0065) followed by a "°" ("combining ring above"; U+030A). Perhaps more surprisingly, the letter sequence "ffi" can be described with a single character "Latin small ligature ffi" with code U+FB03. (One could argue that this is a presentation issue that should not have resulted in different Unicode characters, but we don't make the rules.)

The Unicode standard defines four normalization forms (D, KD,

C, and KC) for strings. See https://www.unicode.org/unicode/reports/tr15/tr15-23.html for the details. In the normalization form C, accented characters are always composed. For example, a sequence of "A" and a combining ring above "°" is combined into a single character "Å". In form D, accented characters are always decomposed into their base letters and combining accents: "Å" is turned into "A" followed by "°". Forms KC and KD also decompose characters such as ligatures or the trademark symbol.

You can choose the degree of normalization that you want a collator to use. The value Collator.NO_DECOMPOSITION does not normalize strings at all. This option is faster, but it might not be appropriate for texts that express characters in multiple forms. The default, Collator.CANONICAL_DECOMPOSITION, uses the normalization form D. This is useful for texts that contain accents but not ligatures. Finally, "full decomposition" uses normalization form KD. See Table 7.9 for examples.

Å ≠ A°			
™ ≠ TM			

Table 7.9: Differences between Decomposition Modes

It is wasteful to have the collator decompose a string many times. If one string is compared against other strings many times, you can save the decomposition in a collation key object. The getCollationKey method returns a CollationKey object that you can use for further, faster comparisons. Here is an example:

```
String a = . . .;
CollationKey aKey = coll.getCollationKey(a);
if(aKey.compareTo(coll.getCollationKey(b)) == 0) // fast comparison
   . . .
```

Finally, you might want to convert strings into their normalized forms even if you don't do collation—for example, to store strings in a database or communicate with another program.

The java.text.Normalizer class carries out the normalization process. Here is an example:

```
String name = "Ångström";
String normalized = Normalizer.normalize(name, Normalizer.Form.N
   // uses normalization form D
```

The normalized string contains ten characters. The “Å” and “ö” are replaced by “A°” and “o¨” sequences.

However, that is not usually the best form for storage and transmission. Normalization form C first applies decomposition and then combines the accents back in a standardized order. According to the W3C, this is the recommended mode for transferring data over the Internet.

The program in Listing 7.3 lets you experiment with collation order. The book companion code also has a GUI version. Type a word into the text field and click the Add button to add it to the list of words. Each time you add another word, or change the locale, strength, or decomposition mode, the list of words is sorted again. An = sign indicates words that are considered identical (see Figure 7.3).

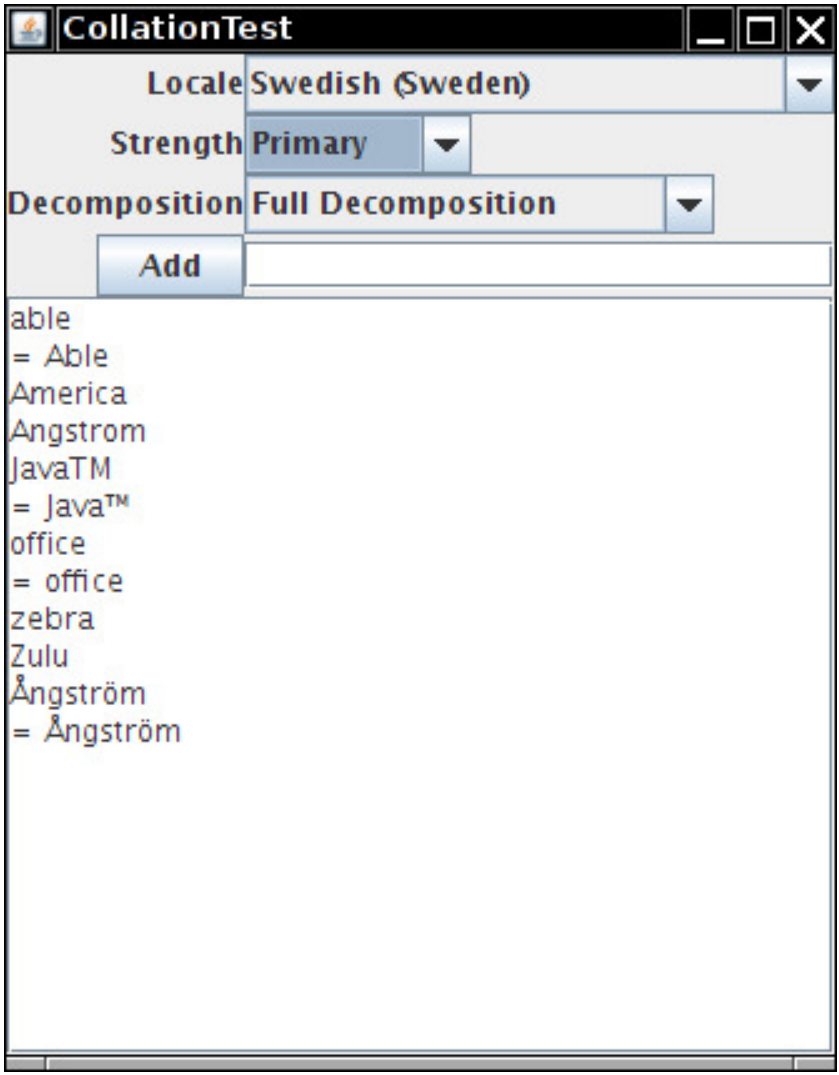


Figure 7.3: The CollationDemo program

The locale names are displayed in sorted order, using the collator of the default locale. If you run this program with the US English locale, note that “Norwegian (Norway,Nynorsk)” comes before “Norwegian (Norway)”. The standard string comparison would have reversed the order since the Unicode value of the comma character is greater than the Unicode value of the closing parenthesis.

Listing 7.3 **v2ch07/collation/CollationDemo2.java**

```
package v2ch07.collation;

import module java.base;
import v2ch07.util.Choices;

/**
 * This program demonstrates collating strings under various locales.
 */
class CollationDemo2 {
  void main() {
    var locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
    Arrays.sort(locales, Comparator.comparing(Locale::getDisplayName));
    Locale locale = Choices.choose(locales, Locale::getDisplayName);

    Collator collator = Collator.getInstance(locale);
    int strength = Choices.choose(Collator.class,
      "Primary", "Secondary", "Tertiary", "Identical");
    int decomposition = Choices.choose(Collator.class,
      "Canonical Decomposition", "Full Decomposition", "No Decomposition");

    List<String> strings = new ArrayList<>();
    strings.add("America");
    strings.add("able");
    strings.add("Zulu");
    strings.add("zebra");
    strings.add("\u00C5ngstr\u00F6m");
    strings.add("A\u030angstro\u0308m");
    strings.add("Angstrom");
    strings.add("Able");
    strings.add("office");
    strings.add("o\uFB03ce");
    strings.add("Java\u2122");
    strings.add("JavaTM");

    collator.setStrength(strength);
    collator.setDecomposition(decomposition);

    strings.sort(collator);

    for (int i = 0; i < strings.size(); i++) {
```

```
      String s = strings.get(i);
      if (i > 0 && collator.compare(s, strings.get(i - 1)) == 0)
        System.out.print("= ");
      System.out.println(s);
    }
  }
}
```

java.text.Collator1.1

- static Locale[] getAvailableLocales()
  returns an array of Locale objects for which Collator objects are available.
- static Collator getInstance()
- static Collator getInstance(Locale l)
  return a collator for the default locale or the given locale.
- int compare(String a, String b)
  returns a negative value if a comes before b, 0 if they are considered identical, and a positive value otherwise.
- boolean equals(String a, String b)
  returns true if a and b are considered identical, false otherwise.
- void setStrength(int strength)
- int getStrength()
  set or get the strength of the collator. Stronger collators tell more words apart. Strength values are Collator.PRIMARY, Collator.SECONDARY, and Collator.TERTIARY.
- void setDecomposition(int decomp)
- int getDecompositon()
  set or get the decomposition mode of the collator. The more a collator decomposes a string, the more strict it will be in deciding whether two strings should be considered identical. Decomposition values are Collator.NO_DECOMPOSITION, Collator.CANONICAL_DECOMPOSITION, and Collator.FULL_DECOMPOSITION.
- CollationKey getCollationKey(String a)
  returns a collation key that contains a decomposition of the characters in a form that can be quickly compared against another collation key.

java.text.CollationKey1.1

- int compareTo(CollationKey b)
  returns a negative value if this key comes before b, 0 if they are considered identical, and a positive value otherwise.

java.text.Normalizer6

- static String normalize(CharSequence str, Normalizer.Form form)
  returns the normalized form of str. The form value is one of NFD, NFKD, NFC, or NFKC.

7.5. Message Formatting

The Java library has a MessageFormat class that formats text with variable parts. It is similar to formatting with the printf method, but it supports locales and formats for numbers and dates. We will examine that mechanism in the following sections.

7.5.1. Formatting Numbers and Dates

Here is a typical message format string:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

The numbers in braces are placeholders for actual names and values. The static method MessageFormat.format lets you substitute values for the variables. It is a “varargs” method, so you can simply supply the parameters as follows:

```
String msg
   = MessageFormat.format("On {2}, a {0} destroyed {1} houses and c
      "hurricane", 99, new GregorianCalendar(1999, 0, 1).getTime(), 10
```

In this example, the placeholder {0} is replaced with "hurricane", {1} is replaced with 99, and so on.

The result of our example is the string

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and caused 100
```

That is a start, but it is not perfect. We don't want to display the time "12:00 AM," and we want the damage amount printed as a currency value. The way we do this is by supplying an optional format for some of the placeholders:

```
"On {2,date,long}, a {0} destroyed {1} houses and caused {3,number,c
```

This example code prints:

```
On January 1, 1999, a hurricane destroyed 99 houses and caused $10
```

In general, the placeholder index can be followed by a type and a style. Separate the index, type, and style by commas. The type can be any of

```
number
time
date
choice
```

If the type is number, then the style can be

```
integer
currency
percent
```

or it can be a DecimalFormat pattern such as $,##0—see Section 7.2.2 for the possible formats.

If the type is either time or date, then the style can be

```
short
medium
long
full
```

or a date format pattern such as yyyy-MM-dd. (See the documentation of the SimpleDateFormat class for the possible formats.)

To include a brace in a message format string, surround it with single quotes, for example: "A {0} is enclosed in '{' and '}' symbols". It is best to use curly apostrophes for text: "{0}’s favorite symbols are braces". But if you must have straight single quotes, you need to double them:

```
String template = "<a href='{0}'>{1}</a>";
```

The static MessageFormat.format method uses the current locale to format the values. To format with an arbitrary locale, you have to work a bit harder because there is no “varargs” method that you can use. You need to place the values to be formatted into an Object[] array, like this:

```
var formatter = new MessageFormat(pattern, locale);
String message = formatter.format(new Object[] { values });
```

java.text.MessageFormat 1.1

- MessageFormat(String pattern)
- MessageFormat(String pattern, Locale locale)
  construct a message format object with the specified pattern and locale.
- void applyPattern(String pattern)
  sets the pattern of a message format object to the specified pattern.
- void setLocale(Locale locale)
- Locale getLocale()
  set or get the locale to be used for the placeholders in the message. The locale is only used for subsequent patterns that you set by calling the applyPattern method.
- static String format(String pattern, Object... args)
  formats the pattern string by using args[i] as input for placeholder {i}.
- StringBuffer format(Object args, StringBuffer result,

FieldPosition pos)
formats the pattern of this MessageFormat. The args parameter must be an array of objects. The formatted string is appended to result, and result is returned. If pos equals new FieldPosition(MessageFormat.Field.ARGUMENT), its beginIndex and endIndex properties are set to the location of the text that replaces the {1} placeholder. Supply null if you are not interested in position information.

java.text.Format1.1

- String format(Object obj)
  formats the given object, according to the rules of this formatter. This method calls format(obj, new StringBuffer(), new FieldPosition(0)).toString().

7.5.2. Choice Formats

Let's look closer at the pattern of the preceding section:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

If we replace the disaster placeholder {0} with "earthquake", the sentence is not grammatically correct in English:

```
On January 1, 1999, a earthquake destroyed . . .
```

What we really want to do is integrate the article "a" into the placeholder:

```
"On {2}, {0} destroyed {1} houses and caused {3} of damage."
```

The {0} would then be replaced with "a hurricane" or "an earthquake". That is especially appropriate if this message needs to be translated into a language where the gender of a word affects the article. For example, in German, the pattern would be

"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden von {3}

The placeholder would then be replaced with the grammatically correct combination of article and noun, such as "Ein Wirbelsturm" or "Eine Naturkatastrophe".

Now let us turn to the {1} parameter. If the disaster wasn't all that catastrophic, {1} might be replaced with the number 1, and the message would read:

On January 1, 1999, a mudslide destroyed 1 houses and . . .

Ideally, we would like the message to vary according to the placeholder value, so it would read

no houses
one house
2 houses
. . .

depending on the placeholder value. The choice formatting option was designed for this purpose.

A choice format is a sequence of pairs, each containing

- A lower limit
- A format string

The lower limit and format string are separated by a # character, and the pairs are separated by | characters.

For example,

{1,choice,0#no houses|1#one house|2#{1} houses}

Table 7.10 shows the effect of this format string for various values of {1}.

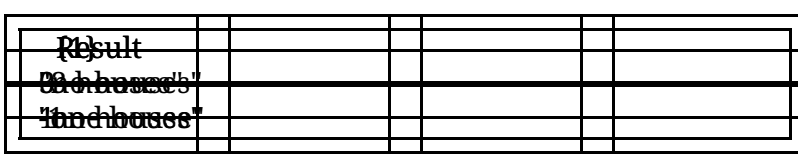


Table 7.10: String Formatted by Choice Format

Why do we use {1} twice in the format string? When the message format applies the choice format to the {1} placeholder and the value is 2, the choice format returns "{1} houses". That string is then formatted again by the message format, and the answer is spliced into the result.

This example shows that the designer of the choice format was a bit muddleheaded. If you have three format strings, you need two limits to separate them. In general, you need one fewer limit than you have format strings. As you saw in Table 7.10, the MessageFormat class ignores the first limit.

The syntax would have been a lot clearer if the designer of this class realized that the limits belong between the choices, such as

```
no houses|1|one house|2|{1} houses // not the actual format
```

You can use the < symbol to denote that a choice should be selected if the lower bound is strictly less than the value.

You can also use the ≤ symbol (U+2264) as a synonym for #. If you like, you can even use the infinity symbol (U+221E) to specify a lower bound of –∞ for the first value.

For example,

```
-∞<no houses|0<one house|2≤{1} houses
```

Let's finish our natural disaster scenario. If we put the choice string inside the original message string, we get the following format instruction:

```
String pattern = """
On {2,date,long}, {0} destroyed {1,choice,0#no houses|1#one house|
and caused {3,number,currency} of damage.
```

```
    """;
```

Or, in German,

```
    String pattern = """
    {0} zerstörte am {2,date,long} {1,choice,0#kein Haus|1#ein Haus|2#
    und richtete einen Schaden von {3,number,currency} an.
    """;
```

Note that the ordering of the words is different in German, but the array of objects you pass to the format method is the same. The order of the placeholders in the format string takes care of the changes in the word ordering.

7.6. Text Boundaries

The BreakIterator class segments text into characters, words, sentences, or lines. The iteration protocol is not very pretty:

```
String s = "A man. A plan. Panama!";
BreakIterator iter = BreakIterator.getSentenceInstance(locale);
iter.setText(s);
int start = iter.first();
int end = iter.next();
while (end != BreakIterator.DONE) {
   String segment = s.substring(start, end);
   start = end;
   end = iter.next();
   IO.println(segment);
}
```

There are four iterators, returned by these static methods:

- The getCharacterInstance method yields an iterator over the grapheme clusters—that is, the Unicode sequences that produce a visible character.
- The getWordInstance method yields an iterator that produces either words or word separators. It is up to you to figure out the type of each segment.
- The iterator of the getSentenceInstance method segments the text into sentences.

- The getLineInstance iterator yields lines.

Each method has a version with no parameter, using the default locale, and one with a Locale parameter.

Instead of iterating over grapheme clusters with a BreakIterator, you can use a regular expression:

```
String[] clusters = s.split("\\b{g}");
```

7.7. Text Input and Output

As you know, the Java programming language itself is fully Unicode-based. However, Windows and Mac OS X still support legacy character encodings such as Windows-1252 or Mac Roman in Western European countries, or Big5 in Taiwan. Therefore, communicating with your users through text is not as simple as it should be. The following sections discuss the complications that you may encounter.

7.7.1. Text Files

Nowadays, it is best to use UTF-8 for saving and loading text files. But you may need to work with legacy files. If you know the expected character encoding, you can specify it when writing or reading text files:

```
Charset windows1252 = Charset.forName("Windows-1252");
var out = new PrintWriter(filename, windows1252);
```

For a guess of the user's preferred encoding, use the “native encoding” that is derived from the system executing the Java virtual machine. As of Java 18, its name is the value of the native.encoding system property.

Prior to Java 18, the native encoding is the default charset, which you can obtain from the static Charset.defaultCharset() method.

7.7.2. Line Endings

This isn't an issue of locales but of platforms. On Windows, text files are expected to use \r\n at the end of each line, where UNIX-based systems only require a \n character. Nowadays, most Windows programs can deal with just a \n, but older programs and the console may give your users grief.

Any line written with the println method will be properly terminated. The only problem is if you print strings that contain \n characters. They are not automatically modified to the platform line ending.

Instead of using \n in strings, you can use printf and the %n format specifier to produce platform-dependent line endings. For example,

```
out.printf("Hello%nWorld%n");
```

produces

```
Hello\r\nWorld\r\n
```

on Windows and

```
Hello\nWorld\n
```

everywhere else.

7.7.3. The Console

If you write programs that communicate with the user through IO, System.in/System.out, or System.console(), you have to face the possibility that the console may use a character encoding that is different from the default charset. This is particularly noticeable when working with the cmd shell on Windows. In the US version of Windows 11, the command shell still uses the archaic IBM437 encoding that originated with IBM personal computers in 1982. The Charset.defaultCharset() method will return the Windows-1252 character set, which is quite different. For example, the euro symbol € is present in

Windows-1252 but not in IBM437. When you call

```
IO.println("100 €");
```

the console will display

```
100 ?
```

Ideally, of course, your users should switch the console to UTF-8. In Windows, the command is

```
chcp 65001
```

As of Java 18, that is sufficient to make Java use UTF-8 in the console. In prior versions, it is also necessary to set the platform encoding with the file.encoding system property:

```
java -Dfile.encoding=UTF-8 MyProgram
```

If you cannot make your users change the character encoding of the console, you can find out what it is. As of Java 18, the system properties stdout.encoding and stderr.encoding give you that information. You can also change the settings on the command line.

You can also call System.console().charset(), provided that System.console() is not null.

> If you obtain user input from a new Scanner(System.in), the scanner uses the default charset and not the console character encoding. If they don't match, this will not work for keyboard input. If System.console() is not null, use a new Scanner(System.console().reader()) instead.

7.7.4. The UTF-8 Byte Order Mark

As already mentioned, it is a good idea to use UTF-8 for text files when you can. However, if your application has to read UTF-8 text files created by other programs, you run into another potential problem. It is perfectly legal to add a “byte

order mark" character U+FEFF as the first character of a file. In the UTF-16 encoding, where each code unit is a two-byte quantity, the byte order mark tells a reader whether the file uses "big-endian" or "little-endian" byte ordering. UTF-8 is a single-byte encoding, so there is no need to specify a byte ordering. But if a file starts with bytes 0xEF 0xBB 0xBF (the UTF-8 encoding of U+FEFF), that is a strong indication that it uses UTF-8. For that reason, the Unicode standard encourages this practice. Any reader is supposed to discard an initial byte order mark.

There is just one fly in the ointment. The Oracle Java implementation stubbornly refuses to follow the Unicode standard, citing potential compatibility issues. That means that you, the programmer, must do what the platform won't do. When you read a text file and encounter a U+FEFF at the beginning, ignore it.

> Unfortunately, the JDK implementors do not follow this advice. When you pass the javac compiler a valid UTF-8 source file that starts with a byte order mark, compilation fails with an error message "illegal character: \65279".

7.7.5. Character Encoding of Source Files

You, the programmer, will need to communicate with the Java compiler—and you do that with tools on your local system. For example, if you use a Chinese or Japanese version of Windows, your Java source code files may be encoded with a local character encoding. Such source code files are not portable. (Only the compiled class files are portable—they use a "modified UTF-8" encoding for identifiers and strings.)

Non-portable source files are not ideal. Nowadays it is easy to configure text editors and development environments to use UTF-8 for source files. As of Java 18, the javac compiler expects UTF-8 encoded source files by default. With earlier versions, run the compiler as:

```
javac -encoding UTF-8 MyProgram.java
```

To compile legacy source files that are not in UTF-8, specify the encoding in the same way.

7.8. Resource Bundles

When localizing an application, you'll probably have a dauntingly large number of message strings, button labels, and so on, that all need to be translated. To make this task feasible, you'll want to define the message strings in an external location, usually called a resource. The person carrying out the translation can then simply edit the resource files without having to touch the source code of the program.

In Java, you can use property files to specify string resources, and you can implement classes for resources of other types.

> Java technology resources are not the same as Windows or Macintosh resources. A Macintosh or Windows executable program stores resources, such as menus, dialog boxes, icons, and messages, in a section separate from the program code. A resource editor can inspect and update these resources without affecting the program code.

> Chapter 5 of Volume I describes a concept of JAR file resources, whereby data files, sounds, and images can be placed in a JAR file. The getResource method of the class Class finds the file, opens it, and returns a URL to the resource. By placing your files into the JAR file, you leave the job of finding the files to the class loader—which already knows how to locate items in a JAR file. However, that mechanism has no locale support.

7.8.1. Locating Resource Bundles

When localizing an application, you produce a set of resource

bundles. Each bundle is a property file or a class that describes locale-specific items (such as messages, labels, and so on). For each bundle, you have to provide versions for all locales that you want to support.

You need to use a specific naming convention for these bundles. For example, resources specific to Germany go into a file *baseName*_de_DE, whereas those shared by all German-speaking countries go into *baseName*_de. In general, use

baseName_language_country

for all country-specific resources, and

baseName_language

for all language-specific resources. Finally, as a fallback, you can put defaults into a file without any suffix.

To load a bundle, use the command

ResourceBundle currentResources = ResourceBundle.getBundle(*base*

The getBundle method attempts to load the bundle that matches the current locale by language and country. If it is not successful, the country and the language are dropped in turn. Then the same search is applied to the default locale, and, finally, the default bundle file is consulted. If even that attempt fails, the method throws a MissingResourceException.

That is, the getBundle method tries to load the following bundles:

baseName_currentLocaleLanguage_currentLocaleCountry
baseName_currentLocaleLanguage
baseName_defaultLocaleLanguage_defaultLocaleCountry
baseName_defaultLocaleLanguage
baseName

Once the getBundle method has located a bundle (say, *baseName*_de_DE), it will still keep looking for *baseName*_de and *baseName*. If these bundles exist, they become the *parents* of the

*baseName*_de_DE bundle in a *resource hierarchy*. Later, when looking up a resource, the parents are searched if a lookup was not successful in the current bundle. That is, if a particular resource was not found in *baseName*_de_DE, then the *baseName*_de and *baseName* will be queried as well.

This is clearly a very useful service—and one that would be tedious to program by hand. The resource bundle mechanism of the Java programming language automatically locates the items that are the best match for a given locale. It is easy to add more and more localizations to an existing program—all you have to do is create additional resource bundles.

If a locale has a script or variant, the lookup is quite a bit more complex. See the documentation of the method ResourceBundle.Control.getCandidateLocales for the gory details.

You need not place all resources for your application into a single bundle. You could have one bundle for button labels, one for error messages, and so on.

7.8.2. Property Files

Internationalizing strings is quite straightforward. You place all your strings into a property file such as MyProgramStrings.properties. This is simply a text file with one key/value pair per line. A typical file would look like this:

```
computeButton=Rechnen
colorName=black
defaultPaperSize=210×297
```

Then you name your property files as described in the preceding section, for example:

```
MyProgramStrings.properties
MyProgramStrings_en.properties
```

MyProgramStrings_de_DE.properties

You can load the bundle simply as

```
ResourceBundle bundle = ResourceBundle.getBundle("MyProgramSt
```

To look up a specific string, call

```
String computeButtonLabel = bundle.getString("computeButton");
```

Property files are encoded as UTF-8.

Before Java 9, files for storing properties had to be ASCII files, and non-ASCII characters had to be encoded using the \uxxxx notation. For example, "colorName=Grün" had to be specified as:

```
colorName=Gr\u00FCn
```

7.8.3. Bundle Classes

To provide resources that are not strings, define classes that extend the ResourceBundle class. Use the standard naming convention to name your classes, for example

MyProgramResources.java
MyProgramResources_en.java
MyProgramResources_de_DE.java

Load the class with the same getBundle method that you use to load a property file:

```
ResourceBundle bundle = ResourceBundle.getBundle("MyProgramR
```

When searching for bundles, a bundle in a class is given preference over a property file when the two bundles have the same base names.

Each resource bundle class implements a lookup table. You

need to provide a key string for each setting you want to localize, and use that key string to retrieve the setting. For example,

```
var backgroundColor = (Color) bundle.getObject("backgroundColor")
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize")
```

The simplest way to implement resource bundle classes is to extend the ListResourceBundle class. The ListResourceBundle lets you place all your resources into an object array and then does the lookup for you. Follow this code outline:

```
public class baseName_language_country extends ListResourceBundle {
   private static final Object[][] contents = {
      { key1, value2 },
      { key2, value2 },
      . . .
   }
   public Object[][] getContents() { return contents; }
}
```

For example,

```
public class ProgramResources_de extends ListResourceBundle {
   private static final Object[][] contents = {
      { "backgroundColor", Color.black },
      { "defaultPaperSize", new double[] { 210, 297 } }
   }
   public Object[][] getContents() { return contents; }
}

public class ProgramResources_en_US extends ListResourceBundle {
   private static final Object[][] contents = {
      { "backgroundColor", Color.blue },
      { "defaultPaperSize", new double[] { 216, 279 } }
   }
   public Object[][] getContents() { return contents; }
}
```

The paper sizes are given in millimeters. Everyone on the

planet, with the exception of the United States and Canada, uses ISO 216 paper sizes. For more information, see https://www.cl.cam.ac.uk/~mgk25/iso-paper.html.

Alternatively, your resource bundle classes can extend the ResourceBundle class. Then you need to implement two methods, to enumerate all keys and to look up the value for a given key:

```
Enumeration<String> getKeys()
Object handleGetObject(String key)
```

The getObject method of the ResourceBundle class calls the handleGetObject method that you supply.

java.util.ResourceBundle 1.1

- static ResourceBundle getBundle(String baseName, Locale locale)
- static ResourceBundle getBundle(String baseName)
  load the resource bundle class with the given name, for the given locale or the default locale, and its parent classes. If the resource bundle classes are located in a package, the base name must contain the full package name, such as "intl.ProgramResources". The resource bundle classes must be public so that the getBundle method can access them.
- Object getObject(String name)
  looks up an object from the resource bundle or its parents.
- String getString(String name)
  looks up an object from the resource bundle or its parents and casts it as a string.
- String[] getStringArray(String name)
  looks up an object from the resource bundle or its parents and casts it as a string array.
- Enumeration<String> getKeys()
  returns an enumeration object to enumerate the keys of this resource bundle. It enumerates the keys in the parent bundles as well.
- Object handleGetObject(String key)

should be overridden to look up the resource value associated with the given key if you define your own resource lookup mechanism.

7.9. A Complete Example

In this section, we apply the material of this chapter to localize a retirement calculator. The program calculates whether or not you are saving enough money for your retirement. You enter your age, how much money you save every month, and so on (see Figure 7.4).

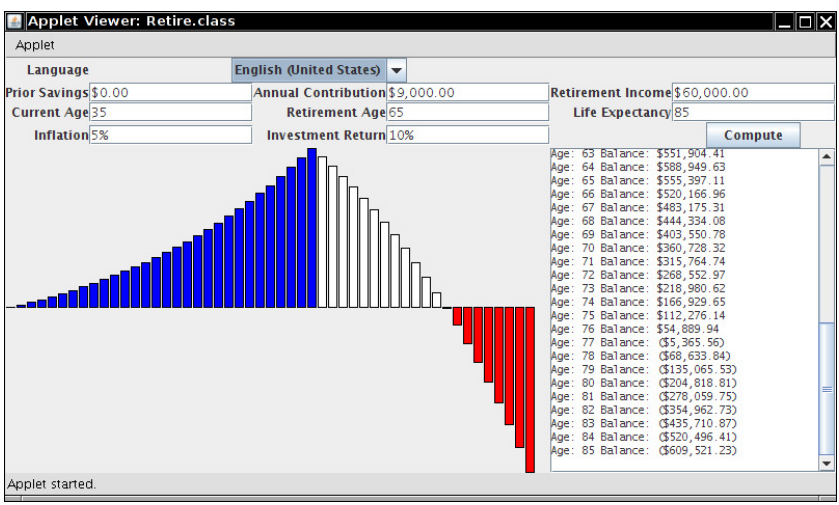


Figure 7.4: The retirement calculator in English

The text area and the graph show the balance of the retirement account for every year. If the numbers turn negative toward the later part of your life and the bars in the graph appear below the *x* axis, you need to do something—for example, save more money, postpone your retirement, die earlier, or be younger.

The retirement calculator works in three locales (English, German, and Chinese). Here are some of the highlights of the

internationalization:

- The labels, buttons, and messages are translated into German and Chinese. You can find them in the classes RetireResources_de and RetireResources_zh. English is used as the fallback—see the RetireResources file.
- Whenever the locale changes, we reset the labels and reformat the contents of the text fields.
- The text fields handle numbers, currency amounts, and percentages in the local format.
- The computation field uses a MessageFormat. The format string is stored in the resource bundle of each language.
- Just to show that it can be done, we use different colors for the bar graph, depending on the language chosen by the user.

Figure 7.5 shows the output in German. To see Chinese characters (as in Figure 7.6), be sure you have Chinese fonts installed and configured with your Java runtime. Otherwise, Chinese characters will show up as “missing character” icons.

Listing 7.4 shows the code of a simplified text version, sadly without colors. Listing 7.5 is the property file for the English strings. The localized strings are in Listing 7.6 (German) and Listing 7.7 (Chinese).

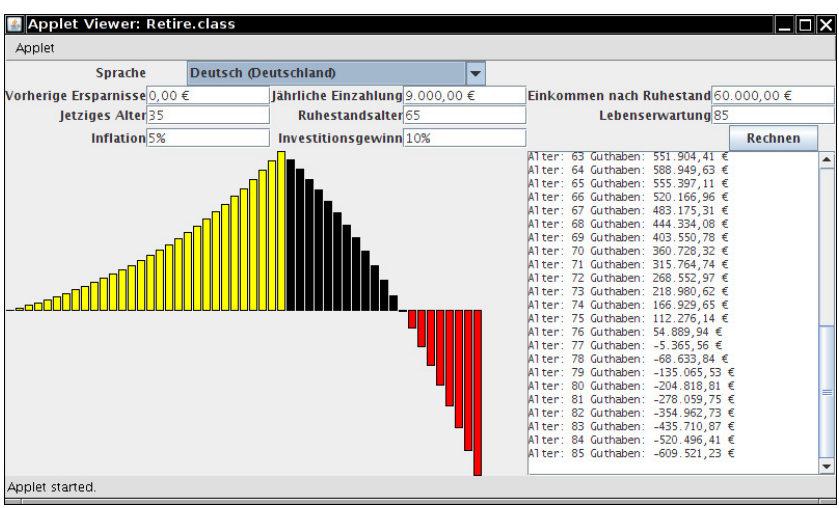


Figure 7.5: The retirement calculator in German

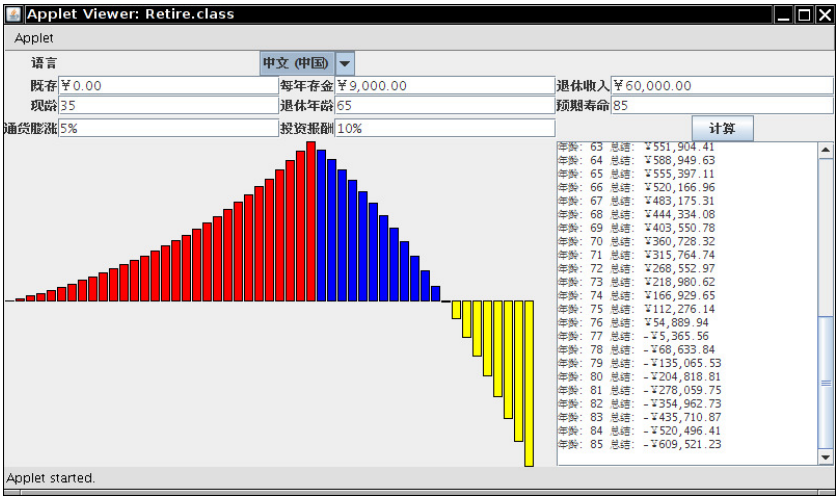


Figure 7.6: The retirement calculator in Chinese

Listing 7.4 **v2ch07/retire2/Retire2.java**

```
package v2ch07.retire2;

import module java.base;

import v2ch07.util.Choices;

/**
 * This program shows a retirement calculator. The prompts are displayed in E
 * Chinese.
 */
class Retire2 {
   void main() throws Exception {
      // Get locale
      ResourceBundle bundle = ResourceBundle.getBundle("v2ch07.retire.Retir
            Locale.getDefault());
      IO.println(bundle.getString("language"));
      Locale[] locales = { Locale.US, Locale.CHINA, Locale.GERMANY };
      Locale currentLocale = Choices.choose(locales, Locale::getDisplayName);
```

```
      bundle = ResourceBundle.getBundle("v2ch07.retire.RetireStrings", curren
      NumberFormat integerFormat = NumberFormat.getIntegerInstance(curre
      NumberFormat numberFormat = NumberFormat.getNumberInstance(cur

      // Prompt for parameters
      double savings = numberFormat.parse(
            IO.readln(bundle.getString("savings") + ": ")).doubleValue();
      double contributions = numberFormat.parse(
            IO.readln(bundle.getString("contrib") + ": ")).doubleValue();
      double income = numberFormat.parse(
            IO.readln(bundle.getString("income") + ": ")).doubleValue();
      int currentAge = integerFormat.parse(
            IO.readln(bundle.getString("currentAge") + ": ")).intValue();
      int retireAge = integerFormat.parse(
            IO.readln(bundle.getString("retireAge") + ": ")).intValue();
      int deathAge = integerFormat.parse(
            IO.readln(bundle.getString("deathAge") + ": ")).intValue();
      double inflationPercent = numberFormat.parse(
            IO.readln(bundle.getString("inflationPercent") + ": ")).doubleValue();
      double investPercent = numberFormat.parse(
            IO.readln(bundle.getString("investPercent") + ": ")).doubleValue();

      // Show values
      var retireMsg = new MessageFormat("");
      retireMsg.setLocale(currentLocale);
      retireMsg.applyPattern(bundle.getString("retire"));

      double balance = savings;
      for (int year = currentAge; year <= deathAge; year++) {
         Object[] msgArgs = { year, balance };
         IO.println(retireMsg.format(msgArgs));
         if (year < retireAge)
            balance += contributions;
         else
            balance -= income;
         balance = balance * (1 + (investPercent - inflationPercent));
      }
   }
}
```

Listing 7.5 **v2ch07/retire2/RetireStrings.properties**

```
language=Language
```

```
computeButton=Compute
savings=Prior Savings
contributions=Annual Contribution
income=Retirement Income
currentAge=Current Age
retireAge=Retirement Age
deathAge=Life Expectancy
inflationPercent=Inflation
investPercent=Investment Return
retire=Age: {0,number} Balance: {1,number,currency}
```

Listing 7.6 **v2ch07/retire2/RetireStrings_de.properties**

```
language=Sprache
computeButton=Rechnen
savings=Vorherige Ersparnisse
contributions=Jährliche Einzahlung
income=Einkommen nach Ruhestand
currentAge=Jetziges Alter
retireAge=Ruhestandsalter
deathAge=Lebenserwartung
inflationPercent=Inflation
investPercent=Investitionsgewinn
retire=Alter: {0,number} Guthaben: {1,number,currency}
```

Listing 7.7 **v2ch07/retire2/RetireStrings_zh.properties**

```
language=语言
computeButton=计算
savings=既存
contributions=每年存金
income=退休收入
currentAge=现龄
retireAge=退休年龄
deathAge=预期寿命
inflationPercent=通货膨涨
investPercent=投资报酬
retire=年龄: {0,number} 总结: {1,number,currency}
```

You have seen how to use the internationalization features of the Java language. You can now use resource bundles to provide translations into multiple languages, and use

formatters and collators for locale-specific text processing.

In the next chapter, we will delve into scripting and compiling.

Chapter 8 ▪ Compiling and Scripting

This chapter introduces two techniques for processing code. You can use the compiler API when you want to compile Java code inside your application. The scripting API lets you invoke code in a scripting language such as JavaScript or Groovy.

8.1. The Compiler API

There are quite a few tools that need to compile Java code. Obviously, development environments and programs that teach Java programming are among them, as well as testing and build automation tools. Another example is the processing of Jakarta Server Pages—web pages with embedded Java statements.

8.1.1. Invoking the Compiler

It is very easy to invoke the compiler. Here is a sample call:

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = . . .;
OutputStream errStream = . . .;
int result = compiler.run(null, outStream, errStream,
   "-sourcepath", "src", "MyProg.java");
```

A result value of 0 indicates successful compilation.

The compiler sends its output and error messages to the provided streams. You can set these arguments to null, in which case System.out and System.err are used. The first argument of the run method is an input stream. As the compiler takes no console input, you can always leave it as null. (The run method is inherited from the generic javax.tools.Tool interface, which allows for tools that read input.)

The remaining arguments of the run method are the arguments that you would pass to javac if you invoked it on the command line. These can be options or file names.

8.1.2. Launching a Compilation Task

You can have more control over the compilation process with a CompilationTask object. This can be useful if you want to supply source from string, capture class files in memory, or process the error and warning messages.

To obtain a CompilationTask, start with a compiler object as in the preceding section. Then call

```
JavaCompiler.CompilationTask task = compiler.getTask(
    errorWriter, // Uses System.err if null
    fileManager, // Uses the standard file manager if null
    diagnosticListener, // Uses System.err if null
    options, // null if no options
    classes, // For annotation processing; null if none
    sources);
```

The last three arguments are Iterable instances. For example, a sequence of options might be specified as

```
Iterable<String> options = List.of("-d", "bin");
```

The sources parameter is an Iterable of JavaFileObject instances. If you want to compile disk files, get a StandardJavaFileManager and call its getJavaFileObjects method:

```
StandardJavaFileManager stdFileManager
   = compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> sources
   = stdFileManager.getJavaFileObjectsFromStrings(List.of("File1.java
JavaCompiler.CompilationTask task
   = compiler.getTask(null, null, null, options, null, sources);
```

> The classes parameter is only used for annotation processing. In that case, you also need to call task.processors(annotationProcessors) with a list of Processor objects. See Chapter 11 of Volume I for an example of annotation processing.

The getTask method returns the task object but does not yet start the compilation process. The CompilationTask class extends Callable<Boolean>. You can submit it to an ExecutorService for concurrent execution, or you can just make a synchronous call:

```
Boolean result = task.call();
```

8.1.3. Capturing Diagnostics

To listen to error messages, install a DiagnosticListener. The listener receives a Diagnostic object whenever the compiler reports a warning or error message. The DiagnosticCollector class implements this interface. It simply collects all diagnostics so that you can iterate through them after the compilation is complete.

```
var collector = new DiagnosticCollector<JavaFileObject>();
compiler.getTask(null, fileManager, collector, null, null, sources).call(
for (Diagnostic<? extends JavaFileObject> d : collector.getDiagnostics
   IO.println(d);
}
```

A Diagnostic object contains information about the problem location (including file name, line number, and column number) as well as a human-readable description.

You can also install a DiagnosticListener to the standard file manager, in case you want to trap messages about missing files:

```
StandardJavaFileManager stdFileManager
   = compiler.getStandardFileManager(diagnosticsListener, null, null
```

8.1.4. Reading Source Files from Memory

If you generate source code on the fly, you can have it compiled from memory, without having to save files to disk. Use this class to hold the code:

```
public class StringSource extends SimpleJavaFileObject {
   private String code;

   StringSource(String name, String code) {
      super(URI.create("string:///" + name.replace(".", "/") + ".java"), Kin
      this.code = code;
   }

   public CharSequence getCharContent(boolean ignoreEncodingErr
      return code;
   }
}
```

Then generate the code for your classes and give the compiler a list of StringSource objects:

```
List<StringSource> sources = List.of(
   new StringSource(className1, class1CodeString),
   new StringSource(className2, class2CodeString),
   . . .);
task = compiler.getTask(null, fileManager, diagnosticsListener, null, n
```

8.1.5. Writing Byte Codes to Memory

If you compile classes on the fly, there is no need to save the class files to disk. You can save them to memory and load them right away.

First, here is a class for holding the bytes:

```
public class ByteArrayClass extends SimpleJavaFileObject {
   private ByteArrayOutputStream out;

   ByteArrayClass(String name) {
      super(URI.create("bytes:///" + name.replace(".", "/") + ".class"), Kin
   }

   public byte[] getCode()  {
      return out.toByteArray();
   }

   public OutputStream openOutputStream() throws IOException {
      out = new ByteArrayOutputStream();
      return out;
   }
}
```

Next, you need to configure the file manager to use these classes for output:

```
var classes = new ArrayList<ByteArrayClass>();
StandardJavaFileManager stdFileManager
   = compiler.getStandardFileManager(null, null, null);
JavaFileManager fileManager
   = new ForwardingJavaFileManager<>(stdFileManager) {
      public JavaFileObject getJavaFileForOutput(Location location,
            String className, Kind kind, FileObject sibling)
            throws IOException {
         if (kind == Kind.CLASS) {
            ByteArrayClass outfile = new ByteArrayClass(className);
            classes.add(outfile);
            return outfile;
         }
         else
            return super.getJavaFileForOutput(location, className, kin
      }
   };
```

To load the classes, you need a class loader (see Chapter 9):

```
public class ByteArrayClassLoader extends ClassLoader {
   private Iterable<ByteArrayClass> classes;

   public ByteArrayClassLoader(Iterable<ByteArrayClass> classes) {
      this.classes = classes;
   }

   public Class<?> findClass(String name) throws ClassNotFoundExce
      for (ByteArrayClass cl : classes) {
         if (cl.getName().equals("/" + name.replace(".", "/") + ".class")) {
            byte[] bytes = cl.getCode();
            return defineClass(name, bytes, 0, bytes.length);
         }
      }
      throw new ClassNotFoundException(name);
   }
}
```

After compilation has finished, call the Class.forName method with that class loader:

```
ByteArrayClassLoader loader = new ByteArrayClassLoader(classes);
Class<?> cl = Class.forName(className, true, loader);
```

8.1.6. An Example: Dynamic Java Code Generation

In the Jakarta Server Pages (JSP) technology for dynamic web pages, you can mix HTML with snippets of Java code, for example:

```
<p>The current date and time is <b><%= LocalDateTime.now() %></
```

The JSP engine dynamically compiles the Java code into a servlet. In our sample application, we use a simpler example and generate code for a very simple worksheet format. Authors include text, Java expressions (enclosed in backticks), and Java statements (enclosed in triple backticks) into a worksheet. A ??? denotes a line of input. The worksheet is compiled into a Java class that prints the text and the values of the expressions, and executes the Java statements. Here is a sample worksheet:

```
What is your name?
```
String name = ???;
```
How old are you?
```
int age = Integer.parseInt(???);
```
Hello, `name`! Next year, you will be `age + 1`.

This is a very minimal text version of a programming notebook (https://docs.jupyter.org/en/latest/#what-is-a-notebook). The sheet is translated into a class whose main method has print statements to emit the text and expression values. The statements are included verbatim. One can envision an enhancement of this program that displays images and tables, just like a real programming notebook.

The buildSource method in the program of Listing 8.1 builds up this code and places it into a StringSource object. That object is passed to the Java compiler.

As described in the preceding section, we use a ForwardingJavaFileManager that constructs a ByteArrayClass object for every compiled class. These objects capture the class file that is generated when the class is compiled. After compilation, we use the class loader from the preceding section to load the class and execute its main method.

This example demonstrates how you can use dynamic compilation with in-memory source and class files.

---

Listing 8.1 **v2ch08/compiler/CompilerDemo.java**

---

```
package v2ch08.compiler;

import module java.base;
import module java.compiler;

import javax.tools.JavaFileObject.Kind;
```
```

```
import javax.tools.ToolProvider;

/**
 * This program demonstrates compiling Java code in a Java program.
 */
public class CompilerDemo {
   void main(String[] args) throws Exception {
      JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

      var classFileObjects = new ArrayList<ByteArrayClass>();

      var diagnosticsListener = new DiagnosticCollector<JavaFileObject>();

      JavaFileManager fileManager
         = compiler.getStandardFileManager(diagnosticsListener, null, null);
      fileManager = new ForwardingJavaFileManager<>(fileManager) {
         public JavaFileObject getJavaFileForOutput(Location location, String cla
               Kind kind, FileObject sibling) throws IOException {
            if (kind == Kind.CLASS) {
               var fileObject = new ByteArrayClass(className);
               classFileObjects.add(fileObject);
               return fileObject;
            }
            else
               return super.getJavaFileForOutput(location, className, kind, siblin
         }
      };

      String mainClassName = "v2ch08.compiler.Main";
      String worksheetName = args.length > 0 ? args[0] : "v2ch08/compiler/work

      StandardJavaFileManager stdFileManager = compiler.getStandardFileMan
            null);
      var sources = new ArrayList<JavaFileObject>();
      for (JavaFileObject file : stdFileManager.getJavaFileObjectsFromStrings(
            List.of(mainClassName.replace(".", "/") + ".java")))
         sources.add(file);

      JavaFileObject source = buildSource(mainClassName, Path.of(worksheetN
      JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager,
            diagnosticsListener, null, null, List.of(source));
      Boolean result = task.call();

      for (Diagnostic<? extends JavaFileObject> d : diagnosticsListener.getDiagn
```

```
            IO.println(d.getKind() + ": " + d.getMessage(null));
        fileManager.close();
        if (!result) {
            IO.println("Compilation failed.");
            System.exit(1);
        }

        var loader = new ByteArrayClassLoader(classFileObjects);
        Class<?> mainClass = loader.loadClass(mainClassName);
        mainClass.getMethod("main").invoke(null);
    }

    /**
     * Builds the source for the class that implements the worksheet actions.
     * @param className the name of the class to be built
     * @param worksheet the file containing the worksheet instructions
     * @return a file object containing the source in a string builder
     */
    JavaFileObject buildSource(String className, Path worksheet) throws IOExc
        var builder = new StringBuilder();
        int n = className.lastIndexOf(".");
        if (n >= 0) {
            String packageName = className.substring(0, n);
            className = className.substring(n + 1);
            builder.append("package ").append(packageName).append(";\n\n");
        }
        builder.append("public class ").append(className).append(" {\n");
        builder.append("public static void main() {\n");
        boolean verbatim = false;
        for (String line : Files.readAllLines(worksheet)) {
            if (line.equals("```"))
                verbatim = !verbatim;
            else if (verbatim)
                builder.append(line.replace("???", "IO.readln()")).append("\n");
            else {
                String[] fragments = line.split("`");
                for (int i = 0; i < fragments.length; i++) {
                    builder.append("IO.print(");
                    if (i % 2 == 0)
                        builder.append("\"").append(
                                fragments[i].replace("\\", "\\\\").replace("\"", "\\\"").
                                .append("\"");
                    else
                        builder.append(fragments[i]);
```

```
                builder.append(");\n");
            }
            builder.append("IO.println()\n;");
        }
    }

    builder.append("} }\n");
    return new StringSource(className, builder.toString());
  }
}
```

javax.tools.Tool6

- int run(InputStream in, OutputStream out, OutputStream err, String... arguments)
  runs the tool with the given input, output, and error streams and the given arguments. Returns 0 for success, a nonzero value for failure.

javax.tools.JavaCompiler6

- StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)
  gets the standard file manager for this compiler. You can supply null for default error reporting, locale, and character set.
- JavaCompiler.CompilationTask getTask(Writer out, JavaFileManager fileManager, DiagnosticListener<? super JavaFileObject> diagnosticListener, Iterable<String> options, Iterable<String> classesForAnnotationProcessing, Iterable<? extends JavaFileObject> sourceFiles)
  gets a compilation task that, when called, will compile the given source files. See the discussion in the preceding section for details.

javax.tools.StandardJavaFileManager6

- Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)
- Iterable<? extends JavaFileObject> getJavaFileObjectsFromPaths(Collection<? extends Path> paths)13
- Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)
  translate a sequence of file names or files into a sequence of JavaFileObject instances.

javax.tools.JavaCompiler.CompilationTask6

- Boolean call()
  performs the compilation task.

javax.tools.DiagnosticCollector<S>6

- DiagnosticCollector()
  constructs an empty collector.
- List<Diagnostic<? extends S>> getDiagnostics()
  gets the collected diagnostics.

javax.tools.Diagnostic<S>6

- S getSource()
  gets the source object associated with this diagnostic.
- Diagnostic.Kind getKind()
  gets the type of this diagnostic—one of ERROR, WARNING, MANDATORY_WARNING, NOTE, or OTHER.
- String getMessage(Locale locale)
  gets the message describing the issue raised in this diagnostic. Pass null for the default locale.
- long getLineNumber()
- long getColumnNumber()
  get the position of the issue raised in this diagnostic.

javax.tools.SimpleJavaFileObject6

- CharSequence getCharContent(boolean ignoreEncodingErrors)
  override this method for a file object that represents a source file and produces the source code.
- OutputStream openOutputStream()
  override this method for a file object that represents a class file and produces a stream to which the bytecodes can be written.

javax.tools.ForwardingJavaFileManager<M extends JavaFileManager>6

- protected ForwardingJavaFileManager(M fileManager)
  constructs a JavaFileManager that delegates all calls to the given file manager.
- FileObject getJavaFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)
  Intercept this call if you want to substitute a file object for writing class files; kind is one of SOURCE, CLASS, HTML, or OTHER.

8.2. Scripting for the Java Platform

A scripting language is a language that avoids the usual edit/compile/link/run cycle by interpreting the program text at runtime. Scripting languages have a number of advantages:

- Rapid turnaround, encouraging experimentation
- Changing the behavior of a running program
- Enabling customization by program users

On the other hand, most scripting languages lack features that are beneficial for programming complex applications, such as

strong typing, encapsulation, and modularity.

It is therefore tempting to combine the advantages of scripting and traditional languages. The scripting API lets you do just that for the Java platform. It enables you to invoke scripts written in JavaScript, Groovy, Ruby, and even exotic languages such as Scheme and Haskell, from a Java program. For example, the Renjin project (https://www.renjin.org) provides a Java implementation of the R programming language, which is commonly used for statistical programming, together with an "engine" of the scripting API.

When looking for a compatible implementation of your favorite language, search for JSR 223 support. (Java Specification Request 223 developed the scripting API before it was integrated into Java.)

The JMeter performance analysis tool (https://jmeter.apache.org/) is an example of a program that allows users to write scripts in any language with Java scripting support. The Maven build tool has a plugin (https://maven.apache.org/plugins/maven-scripting-plugin/index.html) for writing build steps in any language that has a Java scripting engine.

In the following sections, I'll show you how to select an engine for a particular language, how to execute scripts, and how to make use of advanced features that some scripting engines offer.

8.2.1. Getting a Scripting Engine

A scripting engine is a library that can execute scripts in a particular language. When the virtual machine starts, it discovers the available scripting engines. To enumerate them, construct a ScriptEngineManager and invoke the getEngineFactories method. You can ask each engine factory for the supported engine names, MIME types, and file extensions. Table 8.1 shows typical values.

The examples in this chapter use the Rhino scripting engine,

available at https://github.com/mozilla/rhino, that implements an ancient version of JavaScript.

Table 8.1: Properties of Scripting Engine Factories

Usually, you know which engine you need, and you can simply request it by name, MIME type, or extension. For example:

```
var manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("rhino");
```

You need to provide the JAR files that implement the script engine on the classpath.

The Oracle JDK used to contain a JavaScript engine called Nashorn, but it has been removed in Java 15. It is available as a standalone project at https://github.com/openjdk/nashorn.

---

javax.script.ScriptEngineManager6

- List<ScriptEngineFactory> getEngineFactories()
  gets a list of all discovered engine factories.
- ScriptEngine getEngineByName(String name)
- ScriptEngine getEngineByExtension(String extension)
- ScriptEngine getEngineByMimeType(String mimeType)
  get the script engine with the given name, script file extension, or MIME type.

javax.script.ScriptEngineFactory6

- List<String> getNames()
- List<String> getExtensions()
- List<String> getMimeTypes()
  get the names, script file extensions, and MIME types under which this factory is known.

8.2.2. Script Evaluation and Bindings

Once you have an engine, you can call a script simply by invoking

```
Object result = engine.eval(scriptString);
```

If the script is stored in a file, open a Reader and call

```
Object result = engine.eval(reader);
```

You can invoke multiple scripts on the same engine. If one script defines variables, functions, or classes, most scripting engines retain the definitions for later use. For example,

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

will return 1729.

To find out whether it is safe to concurrently execute scripts in multiple threads, call

```
Object param = factory.getParameter("THREADING");
```

The returned value is one of the following:

- null: Concurrent execution is not safe.
- "MULTITHREADED": Concurrent execution is safe. Effects from one thread might be visible from another thread.
- "THREAD-ISOLATED": In addition to "MULTITHREADED", different variable bindings are

maintained for each thread.

- "STATELESS": In addition to "THREAD-ISOLATED", scripts do not alter variable bindings.

You will often want to add variable bindings to the engine. A binding consists of a name and an associated Java object. For example, consider these statements:

```
engine.put("k", 1728);
Object result = engine.eval("k + 1");
```

The script code reads the definition of k from the bindings in the “engine scope.” This is particularly important because most scripting languages can access Java objects, often with a syntax that is simpler than the Java syntax. For example,

```
engine.put("d", LocalDate.now());
Object result = engine.eval("d.month"); // calls getMonth
```

Conversely, you can retrieve variables that were bound by scripting statements:

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

In addition to the engine scope, there is also a global scope. Any bindings that you add to the ScriptEngineManager are visible to all engines.

Instead of adding bindings to the engine or global scope, you can collect them in an object of type Bindings and pass it to the eval method:

```
Bindings scope = engine.createBindings();
scope.put("d", LocalDate.now());
engine.eval(scriptString, scope);
```

This is useful if a set of bindings should not persist for future calls to the eval method.

You might want to have scopes other than the engine and

global scopes. For example, a web container might need request and session scopes. However, then you are on your own. You will need to write a class that implements the ScriptContext interface, managing a collection of scopes. Each scope is identified by an integer number, and scopes with lower numbers should be searched first. (The standard library provides a SimpleScriptContext class, but it only holds global and engine scopes.)

javax.script.ScriptEngine6

- Object eval(String script)
- Object eval(Reader reader)
- Object eval(String script, Bindings bindings)
- Object eval(Reader reader, Bindings bindings)
  evaluate the script given by the string or reader, subject to the given bindings.
- Object get(String key)
- void put(String key, Object value)
  get or put a binding in the engine scope.
- Bindings createBindings()
  creates an empty Bindings object suitable for this engine.

javax.script.ScriptEngineManager6

- Object get(String key)
- void put(String key, Object value)
  get or put a binding in the global scope.

javax.script.Bindings6

- Object get(String key)
- void put(String key, Object value)
  get or put a binding into the scope represented by this Bindings object.

8.2.3. Redirecting Input and Output

You can redirect the standard input and output of a script by calling the setReader and setWriter methods of the script context. For example,

```
var writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

Any output written with the JavaScript print or println functions is sent to writer.

The setReader and setWriter methods only affect the scripting engine's standard input and output sources. For example, if you execute the JavaScript code

```
println("Hello");
java.lang.System.out.println("World");
```

only the first output is redirected.

The Rhino engine does not have the notion of a standard input source. Calling setReader has no effect.

javax.script.ScriptEngine6
■ ScriptContext getContext() gets the default script context for this engine.

javax.script.ScriptContext6
■ Reader getReader() ■ void setReader(Reader reader) ■ Writer getWriter() ■ void setWriter(Writer writer) ■ Writer getErrorWriter() ■ void setErrorWriter(Writer writer) get or set the reader for input or writer for normal or error output.

8.2.4. Calling Scripting Functions and Methods

With some script engines, you can invoke a scripting language function directly in your Java code, without having to evaluate a script code snippet that makes the call. This is useful if you allow users to implement a service in a scripting language of their choice, so that you can call it from Java.

The script engines that offer this functionality implement the Invocable interface. In particular, the Rhino engine implements Invocable.

To call a function, call the invokeFunction method with the function name, followed by the function arguments:

```
// Define greet function in JavaScript
engine.eval("function greet(how, whom) { return how + ', ' + whom +

// Call the function with arguments "Hello", "World"
result = ((Invocable) engine).invokeFunction("greet", "Hello", "World
```

If the scripting language is object-oriented, call invokeMethod:

```
// Define Greeter class in JavaScript
engine.eval("""
function Greeter(how) { this.how = how }
Greeter.prototype.welcome =
   function (whom) { return this.how + ', ' + whom + '!' }
""");

// Construct an instance
Object yo = engine.eval("new Greeter('Yo')");

// Call the welcome method on the instance
result = ((Invocable) engine).invokeMethod(yo, "World");
```

> Rhino does not support the modern JavaScript class syntax. For more information on how to define classes in JavaScript the old-fashioned way, see JavaScript—The

---

> If the script engine does not implement the Invocable interface, you might still be able to call a method in a language-independent way. The getMethodCallSyntax method of the ScriptEngineFactory interface produces a string that you can pass to the eval method. For this, however, all method parameters must be bound to names, whereas invokeMethod can be called with arbitrary values.

You can go a step further and ask the scripting engine to implement a Java interface. Then you can call scripting functions and methods with the Java method call syntax.

The details depend on the scripting engine, but typically you need to supply a function for each method of the interface. For example, consider a Java interface

```
public interface Greeter {
   String welcome(String whom);
}
```

If you define a global function with the same name in Rhino, you can call it through this interface:

```
// Define welcome function in JavaScript
engine.eval("function welcome(whom) { return 'Hello, ' + whom + '!'

// Get a Java object and call a Java method
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
result = g.welcome("World");
```

In an object-oriented scripting language, you can access a script class through a matching Java interface. For example, here is how to call an object of the JavaScript Greeter class with Java syntax:

```
Greeter g = ((Invocable) engine).getInterface(yo, Greeter.class);
result = g.welcome("World");
```

In summary, the Invocable interface is useful if you want to call scripting code from Java without worrying about the scripting language syntax.

javax.script.Invocable6

- Object invokeFunction(String name, Object... args)
- Object invokeMethod(Object thiz, String name, Object... args)
  invoke the function or method with the given name, passing the given arguments.
- <T> T getInterface(Class<T> clasz)
  returns an implementation of the given interface, implementing the methods with functions in the scripting engine.
- <T> T getInterface(Object thiz, Class<T> clasz)
  returns an implementation of the given interface, implementing the methods with the methods of the given object.

8.2.5. Compiling a Script

Some scripting engines can compile scripting code into an intermediate form for efficient execution. Those engines implement the Compilable interface. The following example shows how to compile and evaluate code contained in a script file:

```
if (engine instanceof Compilable compilableEngine) {
   Reader reader = Files.newBufferedReader(path);
   CompiledScript script = compilableEngine.compile(reader);
   script.eval();
}
```

Of course, it only makes sense to compile a script if you need to execute it repeatedly.

javax.script.Compilable[6]

- CompiledScript compile(String script)
- CompiledScript compile(Reader reader)
  compile the script given by a string or reader.

javax.script.CompiledScript[6]

- Object eval()
- Object eval(Bindings bindings)
  evaluate this script.

8.2.6. An Example: Script Sheets

Have a look at the program in Listing 8.2 that executes a sheet in the same format as the example in Section 8.1.6. However, this time, the code is in a scripting language, and it is evaluated when the program runs.

The classpath must contain a scripting engine. Download Rhino and the Rhino engine JAR from https://mvnrepository.com/artifact/org.mozilla/rhino and https://mvnrepository.com/artifact/org.mozilla/rhino-engine and place them into a directory jars. Start the program like this:

```
java -classpath .:v2ch08/script/rhino-1.7.14.jar:v2ch08/script/rhino-e
   v2ch08/script/ScriptDemo.java rhino v2ch08/script/worksheet
```

Listing 8.2 **v2ch08/script/ScriptDemo.java**

```
package v2ch08.script;

import module java.base;
import module java.scripting;

/**
```

```
 * This program demonstrates the Java scripting engine.
 */
class ScriptDemo {
   void main(String[] args) throws Exception {
      var manager = new ScriptEngineManager();
      String language;
      if (args.length == 0) {
         IO.println("Available factories: ");
         for (ScriptEngineFactory factory : manager.getEngineFactories())
            IO.println(factory.getEngineName());
         return;
      }
      else
         language = args[0];

      ScriptEngine engine = manager.getEngineByName(language);
      if (engine == null) {
         System.err.println("No engine for " + language);
         System.exit(1);
      }

      String sheetName = args[1];
      boolean verbatim = false;
      var verbatimCode = new StringBuilder();
      for (String line : Files.readAllLines(Path.of(sheetName))) {
         if (line.equals("```")) {
            verbatim = !verbatim;
            if (!verbatim) {
               engine.eval(verbatimCode.toString());
               verbatimCode.delete(0, verbatimCode.length());
            }
         }
         else if (verbatim)
            verbatimCode.append(line).append("\n");
         else {
            String[] fragments = line.split("`");
            for (int i = 0; i < fragments.length; i++) {
               IO.print(i % 2 == 0 ? fragments[i] : engine.eval(fragments[i]));
            }
            IO.println();
         }
      }
   }
}
```

Listing 8.3 **v2ch08/script/worksheet**

```
var name = java.lang.IO.readln("What is your name? ");
var age = 1 * java.lang.IO.readln("How old are you? ");
```

Hello, `name`! Next year, you will be `age + 1`.

Chapter 9 ■ Security

When Java technology first appeared on the scene, the excitement was not about a well-crafted programming language but about the possibility of safely executing applets delivered over the Internet. Obviously, serving executable applets is only practical when the recipients can be sure the code won't wreak havoc on their machines. Security therefore was and is a major concern of both the designers and the users of Java technology. This means that unlike other languages and systems, where security was implemented as an afterthought or as a reaction to break-ins, security mechanisms are an integral part of Java technology.

The security architecture was composed of three parts:

1. Language and virtual machine design features (bounds checking on arrays, no unchecked type conversions, no pointer arithmetic, and so on).
2. A security manager that controls what the code can do (file access, network access, and so on).
3. Code signing, whereby code authors can use standard cryptographic algorithms to authenticate Java code. Then, the users of the code can determine exactly who created the code and ensure that the code has not been altered after it was signed.

The first part has been a resounding success. C++ programs are routinely vulnerable to attacks such as buffer overruns, but Java provides strong protection. Unfortunately, the other two

parts have not fared as well. The security manager was complex and had a large attack surface. It is being deprecated as of Java 17. The code signing architecture was effectively abandoned with the demise of applets and Java Web Start—two mechanisms for the secure delivery of client-side applications.

In previous editions of this book, the security manager and code signing were the highlights of this chapter, but obviously the need for such coverage has gone away. We will first discuss class loaders that check class files for integrity when they are loaded into the virtual machine. I will demonstrate how that mechanism can detect tampering with class files. Then, you'll learn about an authentication framework and cryptographic algorithms which are useful for many security-related operations.

9.1. Class Loaders

A Java compiler converts source instructions into code for the Java virtual machine. The virtual machine code is stored in a class file with a .class extension. Each class file contains the definition and implementation code for one class or interface. In the following section, you will see how the virtual machine loads these class files.

9.1.1. The Class-Loading Process

The virtual machine loads only those class files that are needed for the execution of a program. For example, suppose program execution starts with MyProgram.class. Here are the steps that the virtual machine carries out:

1. The virtual machine has a mechanism for loading class files—for example, by reading the files from disk or by requesting them from the Web; it uses this mechanism to load the contents of the MyProgram class file.
2. If the MyProgram class has fields or superclasses of another class type, their class files are loaded as well. (The process of loading all the classes that a given class depends on is called resolving the class.)

3. The virtual machine then executes the main method in MyProgram.
4. If the main method or a method that main calls requires additional classes, these are loaded next.

The class loading mechanism doesn't just use a single class loader, however. Every Java program has at least three class loaders:

- The bootstrap class loader
- The platform class loader
- The system class loader (sometimes called the application class loader)

The bootstrap class loader loads the platform classes contained in the modules

```
java.base
java.datatransfer
java.desktop
java.instrument
java.logging
java.management
java.management.rmi
java.naming
java.prefs
java.rmi
java.security.sasl
java.xml
```

as well as a number of JDK-internal modules.

There is no ClassLoader object corresponding to the bootstrap class loader. For example,

```
StringBuilder.class.getClassLoader()
```

returns null.

Prior to Java 9, the Java platform classes were located in a file rt.jar. Nowadays, the Java platform is modular, and each

platform module is contained in a JMOD file (see Chapter 9 of Volume I). The platform class loader loads all classes of the Java platform that are not loaded by the bootstrap class loader.

The system class loader loads application classes from the module path and class path.

Prior to Java 9, an "extension class loader" loaded "standard extensions" from the jre/lib/ext directory, and an "endorsed standards override" mechanism provided a way of overriding certain platform classes (including the CORBA and XML implementations) with newer versions. Both of these mechanisms have been removed.

9.1.2. The Class Loader Hierarchy

Class loaders have a parent/child relationship. Every class loader except for the bootstrap one has a parent class loader. A class loader will give its parent a chance to load any given class and will only load it if the parent has failed. For example, when the system class loader is asked to load a platform class (say, java.lang.StringBuilder), it first asks the platform class loader. That class loader first asks the bootstrap class loader. The bootstrap class loader finds and loads the class, so neither of the other class loaders searches any further.

Some programs have a plugin architecture in which certain parts of the code are packaged as optional plugins. If the plugins are packaged as JAR files, you can simply load the plugin classes with an instance of URLClassLoader.

```
var url = new URL("file:///path/to/plugin.jar");
var pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

Since no parent was specified in the URLClassLoader constructor, the parent of the pluginLoader is the system class loader. Figure 9.1 shows the hierarchy.

.

Figure 9.1: The class loader hierarchy

Prior to Java 9, the system class loader was an instance of URLClassLoader. Some programmers used a cast to access the getURLs method, or added JAR files to the class path by calling the protected addURLs method through reflection. This is no longer possible.

Most of the time, you don't have to worry about the class loader hierarchy. Generally, classes are loaded because they are required by other classes, and that process is transparent to you.

Occasionally, however, you need to intervene and specify a class loader. Consider this example:

- Your application code contains a helper method that calls Class.forName(classNameString).
- That method is called from a plugin class.
- The classNameString specifies a class that is contained in the plugin JAR.

The author of the plugin wants the class to be loaded. However, the helper method's class was loaded by the system class loader, and that is the class loader used by Class.forName. The classes in the plugin JAR are not visible. This phenomenon is called classloader inversion.

To overcome this problem, the helper method needs to use the correct class loader. It can require the class loader as a parameter. Alternatively, it can require that the correct class loader is set as the context class loader of the current thread. This strategy is used by many frameworks (such as JAXP and JNDI).

Each thread has a reference to a class loader, called the context class loader. The main thread's context class loader is the system class loader. When a new thread is created, its context

class loader is set to the creating thread's context class loader. Thus, if you don't do anything, all threads will have their context class loaders set to the system class loader.

However, you can set any class loader by calling

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

The helper method can then retrieve the context class loader:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class<?> cl = loader.loadClass(className);
```

> If you write a method that loads a class by name, it is a good idea to offer the caller the choice between passing an explicit class loader and using the context class loader. Don't simply use the class loader of the method's class.

9.1.3. Using Class Loaders as Namespaces

Every Java programmer knows that package names are used to eliminate name conflicts. There are two classes called Date in the standard library, but of course their real names are java.util.Date and java.sql.Date. The simple name is only a programmer convenience and requires the inclusion of appropriate import statements. In a running program, all class names contain their package names.

It might surprise you, however, that you can have two different classes in the same virtual machine that have the same class and package name. A class is determined by its full name and the class loader. This technique is useful for loading code from multiple sources. For example, an application server uses separate class loaders for each application. This allows the virtual machine to separate classes from different applications, no matter what they are named. Figure 9.2 shows an example. Suppose an application server loads two different applications, and each has a class called Util. Since each class is loaded by a

separate class loader, these classes are entirely distinct and do not conflict with each other.

.

Figure 9.2: Two class loaders load different classes with the same name.

9.1.4. Writing Your Own Class Loader

You can write your own class loader for specialized purposes. That lets you carry out custom checks before you pass the bytecodes to the virtual machine. For example, your class loader may refuse to load a class that has not been marked as "paid for."

To write your own class loader, simply extend the ClassLoader class and override the method

```
findClass(String className)
```

The loadClass method of the ClassLoader superclass takes care of the delegation to the parent and calls findClass only if the class hasn't already been loaded and if the parent class loader was unable to load the class.

Your implementation of this method must do the following:

1. Load the bytecodes for the class from the local file system or some other source.
2. Call the defineClass method of the ClassLoader superclass to present the bytecodes to the virtual machine.

In the program of Listing 9.1, we implement a class loader that loads encrypted class files. The program asks the user for the name of the first class to load (that is, the class containing main) and the decryption key. It then uses a special class loader to load the specified class and calls the main method. The class loader decrypts the specified class and all nonsystem classes that are referenced by it. Finally, the program calls the main

method of the loaded class.

For simplicity, we ignore the 2,000 years of progress in the field of cryptography and use the venerable Caesar cipher for encrypting the class files.

> David Kahn's wonderful book The Codebreakers (Macmillan, 1967, p. 84) refers to Suetonius as a historical source for the Caesar cipher. Caesar shifted the 24 letters of the Roman alphabet by 3 letters, which at the time baffled his adversaries.
>
> When this chapter was first written, the U.S. government restricted the export of strong encryption methods. Therefore, I used Caesar's method for my example because it was clearly legal for export.

Our version of the Caesar cipher has as a key a number between 1 and 255. To decrypt, simply add that key to every byte and reduce modulo 256. The Caesar.java program of Listing 9.2 carries out the encryption.

To not confuse the regular class loader, we use a different extension, .caesar, for the encrypted class files.

To decrypt, the class loader simply subtracts the key from every byte. In the companion code for this book, you will find four class files, encrypted with a key value of 3—the traditional choice. To run the encrypted program, you'll need the custom class loader defined in our ClassLoaderDemo program.

Try out the program by running it with a class name of Calculator and a decryption key of 3. A calculator will pop up. But if you use a different key, the class loader will throw an exception.

Encrypting class files has a number of practical uses (provided, of course, that you use something stronger than the Caesar cipher). Without the decryption key, the class files are useless. They can neither be executed by a standard virtual machine

nor readily disassembled.

This means that you can use a custom class loader to authenticate the user of the class or to ensure that a program has been paid for before it will be allowed to run. Of course, encryption is only one application of a custom class loader. You can use other types of class loaders to solve other problems—for example, storing class files in a database.

Listing 9.1 **v2ch09/classLoader/ClassLoaderDemo.java**

```
package v2ch09.classLoader;

import module java.base;

/**
 * This program demonstrates a custom class loader that decrypts class files.
 */
class ClassLoaderDemo {
   void main(String[] args) throws Exception {
      String className = IO.readln("Class name: ");
      int decryptionKey = Integer.parseInt(IO.readln("Decryption key: "));
      var loader = new CryptoClassLoader(decryptionKey);
      Class<?> c = loader.loadClass(className);
      Method m = c.getMethod("main", String[].class);
      m.invoke(null, (Object) new String[] {});
   }
}

/**
 * This class loader loads encrypted class files.
 */
class CryptoClassLoader extends ClassLoader {
   private int key;

   /**
    * Constructs a crypto class loader.
    * @param k the decryption key
    */
   public CryptoClassLoader(int k) {
      key = k;
   }
```

```
   protected Class<?> findClass(String name) throws ClassNotFoundException {
      try {
         byte[] classBytes = loadClassBytes(name);
         Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
         if (cl == null) throw new ClassNotFoundException(name);
         return cl;
      }
      catch (IOException e) {
         throw new ClassNotFoundException(name);
      }
   }

   /**
    * Loads and decrypt the class file bytes.
    * @param name the class name
    * @return an array with the class file bytes
    */
   private byte[] loadClassBytes(String name) throws IOException {
      String cname = name.replace(".", "/") + ".caesar";
      // Read encrypted class files from the same directory as this class loader
      byte[] bytes = getClass().getResourceAsStream(cname).readAllBytes();
      for (int i = 0; i < bytes.length; i++)
         bytes[i] = (byte) (bytes[i] - key);
      return bytes;
   }
}
```

Listing 9.2 **v2ch09/classLoader/Caesar.java**

```
package v2ch09.classLoader;

import module java.base;

/**
 * Encrypts or decrypts a file using the Caesar cipher. Supply the input file, out
 * and the encryption key (or its negative for decryption) on the command line
 */
class Caesar {
   void main(String[] args) throws Exception {
      try (var in = new FileInputStream(args[0]); var out = new FileOutputStrea
         int key = Integer.parseInt(args[2]);
         boolean done = false;
```

```
            while (!done) {
               int ch = in.read();
               if (ch == -1)
                  done = true;
               else
                  out.write((byte) (ch + key));
            }
         }
      }
}
```

java.lang.Class1.0

- ClassLoader getClassLoader()
  gets the class loader that loaded this class.

java.lang.ClassLoader1.0

- ClassLoader getParent()1.2
  returns the parent class loader, or null if the parent class loader is the bootstrap class loader.
- static ClassLoader getSystemClassLoader()1.2
  gets the system class loader—that is, the class loader that was used to load the first application class.
- protected Class findClass(String name)1.2
  should be overridden by a class loader to find the bytecodes for a class and present them to the virtual machine by calling the defineClass method. In the name of the class, use . as package name separator, and don't use a .class suffix.
- Class defineClass(String name, byte[] byteCodeData, int offset, int length)
  adds a new class to the virtual machine whose bytecodes are provided in the given data range.

java.net.URLClassLoader1.2

- URLClassLoader(URL[] urls)

- URLClassLoader(URL[] urls, ClassLoader parent)
  construct a class loader that loads classes from the given URLs. If a URL ends in a /, it is assumed to be a directory, otherwise it is assumed to be a JAR file.

java.lang.Thread1.0

- ClassLoader getContextClassLoader()1.2
  gets the class loader that the creator of this thread has designated as the most reasonable class loader to use when executing this thread.
- void setContextClassLoader(ClassLoader loader)1.2
  sets a class loader for code in this thread to use for loading classes. If no context class loader is set explicitly when a thread is started, the parent thread's context class loader is used.

9.1.5. Bytecode Verification

When a class loader presents the bytecodes of a newly loaded Java platform class to the virtual machine, these bytecodes are first inspected by a verifier. The verifier checks that the instructions cannot perform actions that are obviously damaging. All classes except for platform classes are verified.

Here are some of the checks that the verifier carries out:

- Variables are initialized before they are used.
- Method calls match the types of object references.
- Rules for accessing private data and methods are not violated.
- Local variable accesses fall within the runtime stack.
- The runtime stack does not overflow.

If any of these checks fails, the class is considered corrupted and will not be loaded.

> If you are familiar with Gödel's theorem, you might wonder how the verifier can prove that a class file is free from type mismatches, uninitialized variables, and stack overflows. Gödel's theorem states that it is impossible to design algorithms that process a program and decide whether it has a particular property (such as being free from stack overflows). Is this a conflict between the public relations department at Oracle and the laws of logic? No—in fact, the verifier is not a decision algorithm in the sense of Gödel. If the verifier accepts a program, it is indeed safe. However, the verifier might reject virtual machine instructions even though they would actually be safe. (You might have run into this issue when you were forced to initialize a variable with a dummy value because the verifier couldn't see that it was going to be properly initialized.)

This strict verification is an important security consideration. Accidental errors, such as uninitialized variables, can easily wreak havoc if they are not caught. More importantly, in the wide open world of the Internet, you must be protected against malicious programmers who create evil effects on purpose. For example, by modifying values on the runtime stack or by writing to the private instance fields of platform objects, a program can break through the security system of a browser.

You might wonder, however, why a special verifier is needed to check all these features. After all, the compiler would never allow you to generate a class file in which an uninitialized variable is used or in which a private instance field is accessed from another class. Indeed, a class file generated by a compiler for the Java programming language always passes verification. However, the bytecode format used in the class files is well documented, and it is an easy matter for someone with experience in assembly programming and a hex editor to manually produce a class file containing valid but unsafe instructions for the Java virtual machine. The verifier is always guarding against maliciously altered class files—not just checking the class files produced by a compiler.

Here's an example of how to construct such an altered class file. We start with the program VerifierDemo.java of Listing 9.3. This is a simple program that calls a method and displays the method's result. The fun method itself just computes 1 + 2.

```
static int fun() {
   int m;
   int n;
   m = 1;
   n = 2;
   int r = m + n;
   return r;
}
```

As an experiment, try to compile the following modification of this program:

```
static int fun() {
   int m = 1;
   int n;
   m = 1;
   m = 2;
   int r = m + n;
   return r;
}
```

Here, n is not initialized, so it could have any random value. Of course, the compiler detects that problem and refuses to compile the program. To create a bad class file, we have to work a little harder. First, run the javap program to find out how the compiler translates the fun method. The command

```
javap -c v2ch09.verifier.VerifierDemo
```

shows the bytecodes in the class file in mnemonic form.

```
Method int fun()
   0 iconst_1
   1 istore_0
   2 iconst_2
   3 istore_1
```

```
4 iload_0
5 iload_1
6 iadd
7 istore_2
8 iload_2
9 ireturn
```

Use a hex editor to change instruction 3 from istore_1 to istore_0. That is, local variable 0 (which is m) is initialized twice, and local variable 1 (which is n) is not initialized at all. We need to know the hexadecimal values for these instructions; these values are readily available from the Java Virtual Machine specification (https://docs.oracle.com/javase/specs/jvms/se21/html/index.html).

```
0 iconst_1 04
1 istore_0 3B
2 iconst_2 05
3 istore_1 3C
4 iload_0  1A
5 iload_1  1B
6 iadd     60
7 istore_2 3D
8 iload_2  1C
9 ireturn  AC
```

You can use any hex editor to carry out the modification. In Figure 9.3, you see the class file VerifierDemo.class loaded into the Gnome hex editor, with the bytecodes of the fun method highlighted.

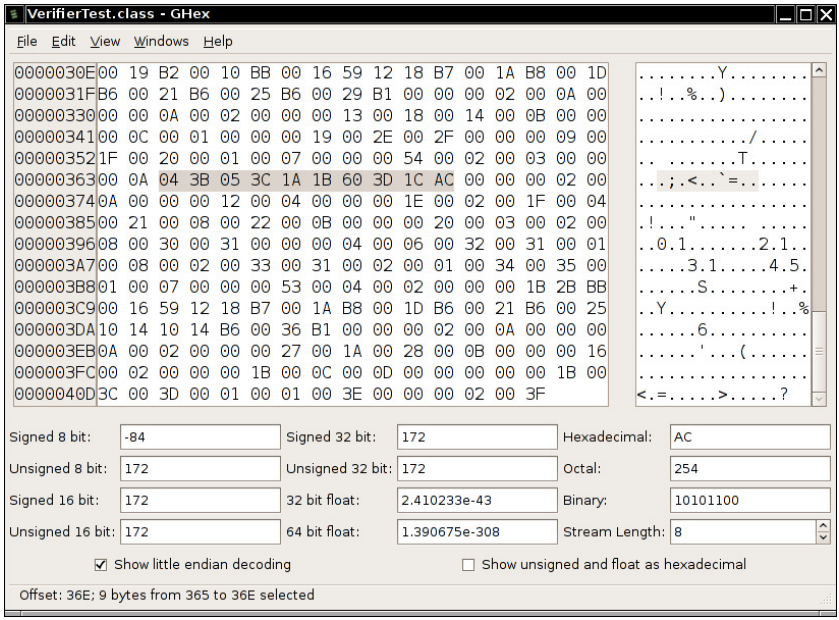


Figure 9.3: Modifying bytecodes with a hex editor

Change 3C to 3B and save the class file. Then try running the VerifierDemo program. You get an error message:

```
Exception in thread "main" java.lang.VerifyError:
    (class: VerifierDemo, method:fun signature: ()I)
    Accessing value from uninitialized register 1
```

That is good—the virtual machine detected our modification.

Now run the program with the -noverify (or -Xverify:none) option:

```
java -noverify v2ch09.verifier.VerifierDemo
```

The fun method returns a seemingly random value. This is actually 2 plus the value that happened to be stored in the variable n, which was never initialized.

Here is a typical printout:

```
1 + 2 == 15102330
```

The -noverify and -Xverify:none options are deprecated and may be removed, but while they are available, they show the impact of the corruption.

Listing 9.3 **v2ch09/verifier/VerifierDemo.java**

```
package v2ch09.verifier;

/**
 * This application demonstrates the bytecode verifier of the virtual machine. I
 * editor to modify the class file, then the virtual machine should detect the tar
 */
class VerifierDemo {
   public static void main(String[] args) {
      System.out.println("1 + 2 == " + fun());
   }

   /**
    * A function that computes 1 + 2.
    * @return 3, if the code has not been corrupted
    */
   public static int fun() {
      int m;
      int n;
      m = 1;
      n = 2;
      // use hex editor to change to "m = 2" in class file
      int r = m + n;
      return r;
   }
}
```

9.2. User Authentication

The Java API provides a framework, called the Java Authentication and Authorization Service (JAAS), that provides access to platform-provided and custom authentication mechanisms. We'll discuss the JAAS framework in the following sections.

9.2.1. The JAAS Framework

As you can tell from its name, the JAAS framework has two components. The “authentication” part is concerned with ascertaining the identity of a program user. The “authorization” part is tied to the deprecated security manager, and we will not discuss it.

JAAS is a “pluggable” API that isolates Java applications from the particular technology used to implement authentication. It supports, among others, UNIX logins, Windows logins, Kerberos authentication, and certificate-based authentication.

Here is the basic outline of the login code:

```
try {
   var context = new LoginContext("Login1"); // defined in JAAS confi
   context.login();
   // get the authenticated Subject
   Subject subject = context.getSubject();
   . . .
   context.logout();
}
catch (LoginException e) { // thrown if login was not successful
   exception.printStackTrace();
}
```

Now the subject denotes the individual who has been authenticated.

The string parameter "Login1" in the LoginContext constructor refers to an entry with the same name in the JAAS configuration file. Here is a sample configuration file:

```
Login1 {
   com.sun.security.auth.module.UnixLoginModule required;
   com.whizzbang.auth.module.RetinaScanModule sufficient debug=
};

Login2 {
   . . .
```

```
};

other {
   . . .
};
```

The module class name and flag can be followed by any number of options that are passed on to the constructor of the module.

The special name other is used when the name of the login context doesn't match any of the names in the configuration.

Of course, the JDK contains no biometric login modules. The following modules are supplied in the com.sun.security.auth.module package:

```
UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule
```

A login policy consists of a sequence of login modules, each labeled required, sufficient, requisite, or optional. The meaning of these keywords is given by the following algorithm:

1. The modules are executed in turn, until a sufficient module succeeds, a requisite module fails, or the end of the module list is reached.
2. Authentication is successful if all required and requisite modules succeed, or if none of them were executed, if at least one sufficient or optional module succeeds.

A login authenticates a subject, which can have multiple principals. A principal describes some property of the subject, such as the username, group ID, or role. The com.sun.security.auth.UnixPrincipal describes the UNIX login name, and the UnixNumericGroupPrincipal can test for membership in a UNIX group.

The program in Listing 9.4 displays the principals of the currently logged-in user. Run the program as

```
java -Djava.security.auth.login.config=v2ch09/auth/jaas.config v2ch0
```

Listing 9.5 shows the login configuration.

On Windows, change UnixLoginModule to NTLoginModule in jaas.config.

Listing 9.4 **v2ch09/auth/AuthDemo.java**

```
package v2ch09.auth;

import module java.base;

/**
 * This program obtains information about a user's Unix login.
 */
class AuthDemo {
   void main() {
      try {
         var context = new LoginContext("Login1");
         context.login();
         IO.println("Authentication successful.");
         Subject subject = context.getSubject();
         for (Principal p : subject.getPrincipals()) {
            IO.println(p.getClass().getName() + ": " + p.getName());
         }
         context.logout();
      }
      catch (LoginException e) {
         IO.println("Authentication failed.");
         e.printStackTrace();
      }
   }
}
```

Listing 9.5 **v2ch09/auth/jaas.config**

```
Login1
```

```
{
  com.sun.security.auth.module.UnixLoginModule required;
};
```

javax.security.auth.login.LoginContext 1.4

- LoginContext(String name)
  constructs a login context. The name corresponds to the login descriptor in the JAAS configuration file.
- void login()
  establishes a login or throws LoginException if the login failed. Invokes the login method on the modules in the JAAS configuration file.
- void logout()
  logs out the subject. Invokes the logout method on the modules in the JAAS configuration file.
- Subject getSubject()
  returns the authenticated subject.

javax.security.auth.Subject 1.4

- Set<Principal> getPrincipals()
  gets the principals of this subject.

java.security.Principal 1.1

- String getName()
  returns the identifying name of this principal.

9.2.2. JAAS Login Modules

In this section, we'll look at a JAAS example that shows you

- How to implement your own login module
- How to implement role-based authorization

Supplying your own login module is useful if you store login information in a database.

One job of the login module is to populate the principal set of the subject that is being authenticated. If a login module supports roles, it adds Principal objects that describe roles. The Java library does not provide a class for this purpose, so I wrote my own (see Listing 9.6).

The login module looks up users, passwords, and roles in a text file that contains lines like this:

```
harry|secret|admin
carl|guessme|HR
```

Of course, in a realistic login module, you would store this information in a database or directory. And you would hash the passwords—see Section 9.3.2.

You can find the code for the SimpleLoginModule in Listing 9.7. The checkLogin method checks whether the username and password match a record in the password file. If so, we add two Principal objects to the subject's principal set:

```
Set<Principal> principals = subject.getPrincipals();
principals.add(new UserPrincipal(username));
principals.add(new RolePrincipal(role));
```

The UserPrincipal class is declared in the com.sun.security.auth package.

The remainder of SimpleLoginModule is straightforward plumbing. The initialize method receives

- The Subject that is being authenticated
- A handler to retrieve login information
- A sharedState map that can be used for communication between login modules
- An options map that contains the name/value pairs that are set in the login configuration

For example, we configure our module as follows:

```
SimpleLoginModule required passwordFile="jaas/password.txt" deb
```

The initialize method retrieves the passwordFile and debug settings from the options parameter.

The login module does not gather the username and password; that is the job of a separate handler. This separation allows you to use the same login module without worrying whether the login information comes from a GUI dialog box, a console prompt, or a configuration file.

The handler is specified when you construct the LoginContext, for example:

```
var context = new LoginContext("Login1",
   new com.sun.security.auth.callback.TextCallbackHandler());
```

The TextCallbackHandler class prompts for username and password from the console.

However, you usually have your own user interface for collecting the username and password. In this case, the callback mechanism is overkill, but there is no other way to transfer that information. The sample application in Listing 9.8 uses a simple handler that merely stores and returns that information.

The handler has a single method, handle, that processes an array of Callback objects. A number of predefined classes, such as NameCallback and PasswordCallback, implement the Callback interface. You could also add your own class, such as RetinaScanCallback. The handler code is a bit unsightly because it needs to analyze the types of the callback objects:

```
public void handle(Callback[] callbacks) {
   for (Callback callback : callbacks) {
      switch (callback) {
         case NameCallback c -> c.setName(username);
         case PasswordCallback c -> c.setPassword(password);
         default -> {}
      }
```

```
    }
  }
```

The login method of the SimpleLoginModule prepares an array of the callbacks that it needs for authentication:

```
var nameCallback = new NameCallback("username: ");
var passwordCallback = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCallback, passwordCa
```

Depending on the handler, the handle method can ask the user or retrieve the information from somewhere. When the handle method returns, the nameCallback and passwordCallback objects have been set. Then the login method retrieves the username and password and checks whether they are valid.

The program in Listing 9.9 uses either the TextCallbackHandler or a custom login, with the username on the command line and a prompt for the password. When the login is successful, the principals are shown. Run the program as

```
java -Djava.security.auth.login.config=v2ch09/jaas/jaas.config v2ch09
```

or

```
java -Djava.security.auth.login.config=v2ch09/jaas/jaas.config v2ch09
```

It is possible to support a more complex two-phase protocol, whereby a login is committed if all modules in the login configuration were successful. For more information, see the login module developer's guide at https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASLMDevGuide.html.

Listing 9.6 **v2ch09/jaas/RolePrincipal.java**

```
package v2ch09.jaas;

import module java.base;
```

```
/**
 * A principal with a named role.
 */
public record RolePrincipal(String name) implements Principal {
   /**
    * @return the role name.
    */
   public String getName() {
      return name;
   }
}
```

Listing 9.7 **v2ch09/jaas/SimpleLoginModule.java**

```
package v2ch09.jaas;

import module java.base;
import com.sun.security.auth.UserPrincipal;

/**
 * This login module authenticates users by reading usernames, passwords, an
 * file.
 */
public class SimpleLoginModule implements LoginModule {
   private Subject subject;
   private CallbackHandler callbackHandler;
   private boolean debug;
   private String passwordFile;

   public void initialize(Subject subject, CallbackHandler callbackHandler,
         Map<String, ?> sharedState, Map<String, ?> options) {
      this.subject = subject;
      this.callbackHandler = callbackHandler;
      debug = options.get("debug").equals("true");
      passwordFile = "" + options.get("passwordFile");
   }

   public boolean login() throws LoginException {
      var nameCallback = new NameCallback("username: ");
      var passwordCallback = new PasswordCallback("password: ", false);
      try {
         if (debug) {
```

```
            System.Logger logger = System.getLogger("com.horstmann.corejava")
            logger.log(System.Logger.Level.INFO, "Invoking handler");
         }
         if (callbackHandler == null) throw new LoginException("no handler");

         callbackHandler.handle(new Callback[] { nameCallback, passwordCallb
         return checkLogin(nameCallback.getName(), passwordCallback.getPass
      }
      catch (UnsupportedCallbackException | IOException e) {
         var e2 = new LoginException();
         e2.initCause(e);
         throw e2;
      }
   }

   /**
    * Checks whether the authentication information is valid. If it is, the subject
    * principals for the user name and role.
    * @param username the user name
    * @param password a character array containing the password
    * @return true if the authentication information is valid
    */
   private boolean checkLogin(String username, char[] password) throws IOEx
      try (var in = new Scanner(Path.of(passwordFile))) {
         while (in.hasNextLine()) {
            String[] inputs = in.nextLine().split("\\|");
            if (inputs[0].equals(username)
                  && Arrays.equals(inputs[1].toCharArray(), password)) {
               String role = inputs[2];
               Set<Principal> principals = subject.getPrincipals();
               principals.add(new UserPrincipal(username));
               principals.add(new RolePrincipal(role));
               return true;
            }
         }
         return false;
      }
   }

   public boolean logout() {
      return true;
   }

   public boolean abort() {
```

```
      return true;
   }

   public boolean commit() {
      return true;
   }
}
```

Listing 9.8 **v2ch09/jaas/SimpleCallbackHandler.java**

```
package v2ch09.jaas;

import module java.base;

/**
 * This simple callback handler presents the given user name and password.
 */
public class SimpleCallbackHandler implements CallbackHandler {
   private String username;
   private char[] password;

   /**
    * Constructs the callback handler.
    * @param username the user name
    * @param password a character array containing the password
    */
   public SimpleCallbackHandler(String username, char[] password) {
      this.username = username;
      this.password = password;
   }

   public void handle(Callback[] callbacks) {
      for (Callback callback : callbacks) {
         switch (callback) {
            case NameCallback c -> c.setName(username);
            case PasswordCallback c -> c.setPassword(password);
            default -> {
            }
         }
      }
   }
}
```

Listing 9.9 **v2ch09/jaas/JAASDemo.java**

```
package v2ch09.jaas;

import module java.base;
import module jdk.security.auth;

/**
 * This program shows how to authenticate a user via a custom login
 */
class JAASDemo {
   void main(String[] args) {
      CallbackHandler handler;
      if (args.length == 0)
         handler = new TextCallbackHandler();
      else {
         // In most apps, you have your own way of reading the username and pa
         // Here, the user provides the user name as a command line argument a
         // password on the console.
         String username = args[0];
         Console console = System.console();
         char[] password = console.readPassword("Password: ");
         handler = new SimpleCallbackHandler(username, password);
      }

      try {
         var context = new LoginContext("Login1", handler);
         context.login();
         Subject subject = context.getSubject();
         for (Principal p : subject.getPrincipals()) {
            IO.println(p.getClass().getName() + " " + p.getName());
         }
         context.logout();
      }
      catch (LoginException e) {
         e.printStackTrace();
         Throwable cause = e.getCause();
         if (cause != null) cause.printStackTrace();
      }
   }
}
```

Listing 9.10 **v2ch09/jaas/jaas.config**

```
Login1
{
  jaas.SimpleLoginModule required passwordFile="jaas/password.txt" debug=
};
```

javax.security.auth.callback.CallbackHandler 1.4

- void handle(Callback[] callbacks)
  handles the given callbacks, interacting with the user if desired, and stores the security information in the callback objects.

javax.security.auth.callback.NameCallback 1.4

- NameCallback(String prompt)
- NameCallback(String prompt, String defaultName)
  construct a NameCallback with the given prompt and default name.
- String getName()
- void setName(String name)
  get or set the name gathered by this callback.
- String getPrompt()
  gets the prompt to use when querying this name.
- String getDefaultName()
  gets the default name to use when querying this name.

javax.security.auth.callback.PasswordCallback 1.4

- PasswordCallback(String prompt, boolean echoOn)
  constructs a PasswordCallback with the given prompt and a flag indicating whether the password should be displayed as it is typed.
- char[] getPassword()
- void setPassword(char[] password)
  get or set the password gathered by this callback.
- String getPrompt()
  gets the prompt to use when querying this password.
- boolean isEchoOn()

gets the echo flag to use when querying this password.

javax.security.auth.spi.LoginModule 1.4

- void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)
  initializes this LoginModule for authenticating the given subject. During login processing, uses the given handler to gather login information. Use the sharedState map for communicating with other login modules. The options map contains the name/value pairs specified in the login configuration for this module instance.
- boolean login()
  carries out the authentication process and populates the subject's principals. Returns true if the login was successful.
- boolean commit()
  is called after all login modules were successful, for login scenarios that require a two-phase commit. Returns true if the operation was successful.
- boolean abort()
  is called if the failure of another login module caused the login process to abort. Returns true if the operation was successful.
- boolean logout()
  logs out this subject. Returns true if the operation was successful.

9.3. Digital Signatures

As already mentioned, applets were what started the Java craze. In practice, people discovered that although they could write animated applets (like the famous "nervous text"), applets could not do a whole lot of useful stuff in the JDK 1.0 security model. For example, since applets under JDK 1.0 were so closely supervised, they couldn't do much good on a corporate intranet, even though relatively little risk attaches to

executing an applet from your company's secure intranet. It quickly became clear to Sun that for applets to become truly useful, users need to be able to assign different levels of security, depending on where the applet originated. If an applet comes from a trusted supplier and has not been tampered with, the user of that applet can decide whether to give the applet more privileges.

To give more trust to an applet, we need to know two things:

- Where did the applet come from?
- Was the code corrupted in transit?

In the past 50 years, mathematicians and computer scientists have developed sophisticated algorithms for ensuring the integrity of data and for creating electronic signatures. The java.security package contains implementations of many of these algorithms. Fortunately, you don't need to understand the underlying mathematics to use the algorithms in the java.security package. In the next sections, I'll show you how message digests can detect changes in data files and how digital signatures can prove the identity of the signer.

9.3.1. Message Digests

A message digest is a digital fingerprint of a block of data. For example, the so-called SHA-1 (Secure Hash Algorithm #1) condenses any data block, no matter how long, into a sequence of 160 bits (20 bytes). As with real fingerprints, one hopes that no two different messages have the same SHA-1 fingerprint. Of course, that cannot be true—there are only 2^{160} SHA-1 fingerprints, so there must be some messages with the same fingerprint. But 2^{160} is so large that the probability of a collision is negligible. How negligible? According to James Walsh in True Odds: How Risks Affect Your Everyday Life (Merritt Publishing, 1996), the chance that you will die from being struck by lightning is about one in 30,000. Now, think of nine other people—for example, your nine least favorite managers or professors. The chance that you and all of them will die from lightning strikes is higher than that of a forged

message having the same SHA-1 fingerprint as the original. (Of course, more than ten people, none of whom you are likely to know, will die from lightning strikes. However, we are talking about the far slimmer chance that your particular choice of people will be wiped out.)

A message digest has two essential properties:

- If one bit or several bits of the data are changed, the message digest also changes.
- A forger who is in possession of a given message cannot construct a fake message that has the same message digest as the original.

The second property is, again, a matter of probabilities. Consider the following message by the billionaire father:

> "Upon my death, my property shall be divided equally among my children; however, my son George shall receive nothing."

That message (with a final newline) has an SHA-1 fingerprint of

12 5F 09 03 E7 31 30 19 2E A6 E7 E4 90 43 84 B4 38 99 8F 67

The distrustful father has deposited the message with one attorney and the fingerprint with another. Now, suppose George bribes the lawyer holding the message. He wants to change the message so that Bill gets nothing. Of course, that changes the fingerprint to a completely different bit pattern:

7D F6 AB 08 EB 40 EC CD AB 74 ED E9 86 F9 ED 99 D1 45 B1 57

Can George find some other wording that matches the fingerprint? If he had been the proud owner of a billion computers from the time the Earth was formed, each computing a million messages a second, he would not yet have found a message he could substitute.

A number of algorithms have been designed to compute such message digests. Among them are SHA-1, the secure hash algorithm developed by the National Institute of Standards and

Technology, and MD5, an algorithm invented by Ronald Rivest of MIT. Both algorithms scramble the bits of a message in ingenious ways. For details about these algorithms, see, for example, Cryptography and Network Security, Seventh Edition, by William Stallings (Pearson, 2017). However, subtle regularities have been discovered in both algorithms, and NIST recommends to switch to stronger alternatives. Java supports the SHA-2 and SHA-3 sets of algorithms.

The MessageDigest class is a factory for creating objects that encapsulate the fingerprinting algorithms. It has a static method, called getInstance, that returns an object of a class that extends the MessageDigest class. This means the MessageDigest class serves double duty:

- As a factory class
- As the superclass for all message digest algorithms

For example, here is how you obtain an object that can compute SHA fingerprints:

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

After you have obtained a MessageDigest object, feed it all the bytes in the message by repeatedly calling the update method. For example, the following code passes all bytes in a file to the alg object just created to do the fingerprinting:

```
InputStream in = . . .;
int ch;
while ((ch = in.read()) != -1)
   alg.update((byte) ch);
```

Alternatively, if you have the bytes in an array, you can update the entire array at once:

```
byte[] bytes = . . .;
alg.update(bytes);
```

When you are done, call the digest method. This method pads the input as required by the fingerprinting algorithm, does the

computation, and returns the digest as an array of bytes.

```
byte[] hash = alg.digest();
```

The program in Listing 9.11 computes a message digest. You can specify the file and algorithm on the command line:

```
java v2ch09.hash.Digest v2ch09/hash/input.txt SHA-1
```

If you do not supply command-line arguments, you will be prompted for the file and algorithm name.

Listing 9.11 **v2ch09/hash/Digest.java**

```
package v2ch09.hash;

import module java.base;

/**
 * This program computes the message digest of a file.
 */
class Digest {
   /**
    * @param args args[0] is the filename, args[1] is optionally the algorithm
    * (SHA-1, SHA-256, or MD5)
    */
   void main(String[] args) throws Exception {
      String filename = args.length >= 1 ? args[0] : IO.readln("File name: ");
      String algname;
      if (args.length >= 2)
         algname = args[1];
      else {
         IO.println("Select one of the following algorithms: ");
         for (Provider p : Security.getProviders())
            for (Provider.Service s : p.getServices())
               if (s.getType().equals("MessageDigest")) IO.println(s.getAlgorithm())
         algname = IO.readln("Algorithm: ");
      }
      MessageDigest alg = MessageDigest.getInstance(algname);
      byte[] input = Files.readAllBytes(Path.of(filename));
      byte[] hash = alg.digest(input);
      for (byte h : hash)
```

```
            IO.print("%02X ".formatted(h & 0xFF));
        IO.println();
    }
}
```

java.security.MessageDigest1.1

- static MessageDigest getInstance(String algorithmName)
  returns a MessageDigest object that implements the specified algorithm. Throws NoSuchAlgorithmException if the algorithm is not provided.
- void update(byte input)
- void update(byte[] input)
- void update(byte[] input, int offset, int len)
  update the digest, using the specified bytes.
- byte[] digest()
  completes the hash computation, returns the computed digest, and resets the algorithm object.
- void reset()
  resets the digest.

9.3.2. Password Hashing

For obvious security reasons, one should never store passwords as plain text. Instead, their hashed values should be stored. When a user logs in, the user's password entry should also be hashed. If the values match, the user is authenticated.

However, the hash algorithms that you have seen in the preceding section are not suitable for password hashing. In order to thwart brute force attacks, the hashing mechanism should use many rounds of computations. Also, to avoid precomputing of password tables, each password should be modified with a random “salt” value.

Java provides the PBKDF2 algorithm for password hashing. The algorithm has four parameters:

- A char array holding the password

- An array of random salt bytes
- The number of iterations
- The number of bits for the output

A salted password isn't a message, so you don't use the MessageDigest interface. Instead, use a SecretKeyFactory like this:

```
var spec = new PBEKeySpec(password, salt, ITERATIONS, 8 * HASH_L
SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithH
byte[] hash = skf.generateSecret(spec).getEncoded();
```

The acronym PBE denotes “password based encryption” because the derived bytes could also be used as an encryption key. PBKDF is a “password based key derivation function”, which turns a short password into a longer encryption key. We are not interested in encryption and just store the resulting bytes. For password verification, it is also important to store the salt. Since it is good practice to increase the number of iterations over time, as processors beccome more powerful, you should store that value as well.

When storing the hash as a string, the bytes should be encoded in hex or, more efficiently, in base64:

```
Base64.Encoder encoder = Base64.getEncoder();
String hashedPassword = ITERATIONS + "|" + encoder.encodeToStrin
   + encoder.encodeToString(hash);
```

To verify a user-entered password, obtain the iterations and salt from the stored hash, and recompute the hash.

9.3.3. Message Signing

In Section 9.3.1, you saw how to compute a message digest—a fingerprint for a message. If the message is altered, the fingerprint of the altered message will not match the fingerprint of the original. If the message and its fingerprint are delivered separately, the recipient can check whether the message has been tampered with. However, if both the message and the fingerprint were intercepted, it is an easy matter to

modify the message and then recompute the fingerprint. After all, the message digest algorithms are publicly known, and they don't require secret keys. In that case, the recipient of the forged message and the recomputed fingerprint would never know that the message has been altered. Digital signatures solve this problem.

To help you understand how digital signatures work, let's look at a few concepts from the field called public key cryptography. Public key cryptography is based on the notion of a public key and private key. The idea is that you tell everyone in the world your public key. However, only you hold the private key, and it is important that you safeguard it and don't release it to anyone else. The keys are matched by mathematical relationships, though the exact nature of these relationships is not important to us. (If you are interested, look it up in The Handbook of Applied Cryptography at https://cacr.uwaterloo.ca/hac/.)

The keys are quite long and complex. For example, here is a matching pair of public and private Digital Signature Algorithm (DSA) keys.

Public key:

p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a
ed0899bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71
935630e1c2062354d0da20a6c416e50be794ca4

y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2d2a8123ce5a8018
40b927281ddb22cb9bc4df596d7de4d1b977d50

Private key:

p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a
ed0899bcd132acd50d99151bdc43ee737592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71
935630e1c2062354d0da20a6c416e50be794ca4

x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a

It is believed to be practically impossible to compute one key from the other. That is, even though everyone knows your public key, they can't, in your lifetime, compute your private key, no matter how many computing resources they have available.

It may be difficult to believe that the private key can't be computed from the public key—but nobody has found an algorithm to do this for the encryption algorithms in common use today. If the keys are sufficiently long, brute force—simply trying all possible keys—would require more computers than can be built from all the atoms in the solar system, crunching away for thousands of years. Of course it is possible that someone could come up with algorithms for computing keys that are much more clever than brute force. For example, the RSA algorithm (the encryption algorithm invented by Rivest, Shamir, and Adleman) depends on the difficulty of factoring large numbers. For the last 20 years, many of the best mathematicians have tried to come up with good factoring algorithms, but so far with no success. For that reason, most cryptographers believe that keys with a "modulus" of 2,000 bits or more are currently completely safe from any attack. DSA is believed to be similarly secure.

Figure 9.4 illustrates how the process works in practice.

.

Figure 9.4: Public key signature exchange with DSA

Suppose Alice wants to send Bob a message, and Bob wants to know this message came from Alice and not an impostor. Alice writes the message and signs the message digest with her private key. Bob gets a copy of her public key. Bob then applies the public key to verify the signature. If the verification passes, Bob can be assured of two facts:

- The original message has not been altered.

- The message was signed by Alice, the holder of the private key that matches the public key that Bob used for verification.

You can see why the security of private keys is so important. If someone steals Alice's private key, or if a government can require her to turn it over, then she is in trouble. The thief or a government agent can now impersonate her by sending messages, such as money transfer instructions, that others will believe came from Alice.

9.3.4. Verifying a Signature

The JDK comes with the keytool program, which is a command-line tool to generate and manage a set of certificates. We expect that ultimately the functionality of this tool will be embedded in other, more user-friendly programs. But right now, we'll use keytool to show how Alice can sign a document and send it to Bob, and how Bob can verify that the document really was signed by Alice and not an impostor.

The keytool program manages keystores—databases of certificates and private/public key pairs. Each entry in the keystore has an alias. Here is how Alice creates a keystore, alice.jks, and generates a key pair with alias alice:

```
keytool -genkeypair -keyalg DSA -keystore alice.jks -alias alice
```

When creating or opening a keystore, you are prompted for a keystore password. For this example, just use secret. If you were to use the keytool-generated keystore for any serious purpose, you would need to choose a good password and safeguard this file.

When generating a key, you are prompted for the following information:

```
Enter keystore password:  secret
Reenter new password:  secret
What is your first and last name?
  [Unknown]:  Alice Lee
```

```
What is the name of your organizational unit?
  [Unknown]:  Engineering
What is the name of your organization?
  [Unknown]:  ACME Software
What is the name of your City or Locality?
  [Unknown]:  San Francisco
What is the name of your State or Province?
  [Unknown]:  CA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Francis
  [no]:  yes
```

The keytool uses names in the X.500 format, whose components are Common Name (CN), Organizational Unit (OU), Organization (O), Location (L), State (ST), and Country (C), to identify key owners and certificate issuers.

Suppose Alice wants to give her public key to Bob. She needs to export a certificate file:

```
keytool -exportcert -keystore alice.jks -alias alice -file alice.cer
```

Now Alice can send the certificate to Bob. When Bob receives the certificate, he can print it:

```
keytool -printcert -file alice.cer
```

The printout looks like this:

```
Owner: CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Francisco, ST=CA, C=
Issuer: CN=Alice Lee, OU=Engineering, O=ACME Software, L=San Francisco, ST=CA, C=
Serial number: b2b46ccd99479da2
Valid from: Tue Jan 16 08:11:22 CET 2024 until: Mon Apr 15 09:11:22 CEST 2024
Certificate fingerprints:
  SHA1: 41:38:E6:F2:9D:FA:90:FE:CB:36:85:F1:E6:39:62:A1:5A:DD:16:BC
  SHA256: 14:CC:AD:BC:83:D5:0C:F9:98:24:52:29:B3:AF:B2:B9:05:A7:C0:61:15:63:EA:FB:1
Signature algorithm name: SHA256withDSA
Subject Public Key Algorithm: 2048-bit DSA key
Version: 3
```

If Bob wants to check that he got the right certificate, he can call Alice and verify the certificate fingerprint over the phone.

Some certificate issuers publish certificate fingerprints on their web sites. For example, to check the DigiCert certificate in the keystore var jdk/lib/security/cacerts

directory, use the -list option:

```
keytool -list -v -keystore $JAVA_HOME/lib/security/cacerts
```

The password for this keystore is changeit. One of the certificates in this keystore is

```
Owner: CN=DigiCert Assured ID Root G3, OU=www.digicert.com, O=I
Issuer: CN=DigiCert Assured ID Root G3, OU=www.digicert.com, O=D
Serial number: ba15afa1ddfa0b54944afcd24a06cec
Valid from: Thu Aug 01 14:00:00 CEST 2013 until: Fri Jan 15 13:00:00
Certificate fingerprints:
     SHA1: F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:3
     SHA256: 7E:37:CB:8B:4C:47:09:0C:AB:36:55:1B:A6:F4:5D:B8:40:68
          16:6A:95:2D:B1:00:71:7F:43:05:3F:C2
```

You can check that your certificate is valid by visiting the web site https://www.digicert.com/kb/digicert-root-certificates.htm.

Once Bob trusts the certificate, he can import it into his keystore.

```
keytool -importcert -keystore bob.jks -alias alice -file alice.cer
```

Never import into a keystore a certificate that you don't fully trust. Once a certificate is added to the keystore, any program that uses the keystore assumes that the certificate can be used to verify signatures.

Now Alice can start sending signed documents to Bob. The jarsigner tool signs and verifies JAR files. Alice simply adds the document to be signed into a JAR file.

```
jar cvf document.jar document.txt
```

She then uses the jarsigner tool to add the signature to the file.

She needs to specify the keystore, the JAR file, and the alias of the key to use.

```
jarsigner -keystore alice.jks document.jar alice
```

When Bob receives the file, he uses the -verify option of the jarsigner program.

```
jarsigner -verify -keystore bob.jks document.jar
```

Bob does not need to specify the key alias. The jarsigner program finds the X.500 name of the key owner in the digital signature and looks for a matching certificate in the keystore.

If the JAR file is not corrupted and the signature matches, the jarsigner program prints

```
jar verified.
```

followed by a number of warnings that you can ignore for this simple experiment. Otherwise, the program displays an error message.

> The KeyStore Explorer program (https://keystore-explorer.org/) provides a graphical user interface for working with keystores.

9.3.5. The Authentication Problem

Suppose you get a message from your friend Alice, signed with her private key, using the method I just showed you. You might already have her public key, or you can easily get it by asking her for a copy or by getting it from her web page. Then, you can verify that the message was in fact authored by Alice and has not been tampered with. Now, suppose you get a message from a stranger who claims to represent a famous software company, urging you to run a program attached to the message. The stranger even sends you a copy of his public key so you can verify that he authored the message. You check that the signature is valid. This proves that the message was signed with

the matching private key and has not been corrupted.

Be careful: You still have no idea who wrote the message. Anyone can generate a pair of public and private keys, sign the message with the private key, and send the signed message and the public key to you. The problem of determining the identity of the sender is called the authentication problem.

The usual way to solve the authentication problem is simple. Suppose the stranger and you have a common acquaintance you both trust. Suppose the stranger meets your acquaintance in person and hands over a USB stick with the public key. Your acquaintance later meets you, assures you that he met the stranger and that the stranger indeed works for the famous software company, and then gives you the USB stick (see Figure 9.5). That way, your acquaintance vouches for the authenticity of the stranger.

.

Figure 9.5: Authentication through a trusted intermediary

In fact, your acquaintance does not actually need to meet you. Instead, he can use his private key to sign the stranger's public key file (see Figure 9.6).

.

Figure 9.6: Authentication through a trusted intermediary's signature

When you get the public key file, you verify the signature of your friend, and because you trust him, you are confident that he did check the stranger's credentials before applying his signature.

However, you might not have a common acquaintance. Some trust models assume that there is always a "chain of trust"—a chain of mutual acquaintances—so that you trust every

member of that chain. In practice, of course, that isn't always true. You might trust your friend, Alice, and you know that Alice trusts Bob, but you don't know Bob and aren't sure that you trust him. Other trust models assume that there is a benevolent big brother—a company in which we all trust. Companies with confidence-inspiring names such as DigiCert, GlobalSign, and Entrust provide verification services.

You will often encounter digital signatures signed by one or more entities who will vouch for the authenticity, and you will need to evaluate to what degree you trust the authenticators. You might place a great deal of trust in a particular certificate authority, perhaps because you saw their logo on many web pages or because you heard that they require multiple people with black attaché cases to come together into a secure chamber whenever new master keys are to be minted.

However, you should have realistic expectations about what is actually being authenticated. You can get a "class 1" ID simply by filling out a web form and paying a small fee. The key is mailed to the e-mail address included in the certificate. Thus, you can be reasonably assured that the e-mail address is genuine, but the requestor could have filled in any name and organization. There are more stringent classes of IDs. For example, with a "class 3" ID, the certificate authority will require an individual requestor to appear before a notary public, and it will check the financial rating of a corporate requestor. Other authenticators will have different procedures. Thus, when you receive an authenticated message, it is important that you understand what, in fact, is being authenticated.

9.3.6. Certificate Signing

In Section 9.3.4 you saw how Alice used a certificate to distribute a public key to Bob. However, Bob needed to ensure that the certificate was valid by verifying the fingerprint with Alice.

Suppose Alice wants to send her colleague Cindy a signed

message, but Cindy doesn't want to bother with verifying lots of signature fingerprints. Now suppose there is an entity that Cindy trusts to verify signatures. In this example, Cindy trusts the Information Resources Department at ACME Software.

That department operates a certificate authority (CA). Everyone at ACME has the CA's public key in their keystore, installed by a system administrator who carefully checked the key fingerprint. The CA signs the keys of ACME employees. When they install each other's keys, the keystore will trust them implicitly because they are signed by a trusted key.

Here is how you can simulate this process. Create a keystore acmesoft.jks. Generate a key pair and export the public key:

```
keytool -genkeypair -keystore acmesoft.jks -keyalg DSA -alias acmero
keytool -exportcert -keystore acmesoft.jks -alias acmeroot -file acmer
```

The public key is exported into a "self-signed" certificate. Then, add it to every employee's keystore:

```
keytool -importcert -keystore cindy.jks -alias acmeroot -file acmeroot
```

For Alice to send messages to Cindy and to everyone else at ACME Software, she needs to bring a signing request for her certificate to the Information Resources Department, and have her key signed. Here are the steps:

```
keytool -keystore alice.jks -alias alice -certreq -file alice.req
keytool -storepass secret -keystore acmesoft.jks -gencert -alias acmer
   -infile alice.req -outfile alice_signed_by_acmeroot.cer
```

Alice gives the file alice_signedby_acmeroot.cer to Cindy and to anyone else in ACME Software. Alternatively, ACME Software can simply store the file in a company directory. Remember, this file contains Alice's public key and an assertion by ACME Software that this key really belongs to Alice.

Now Cindy imports the signed certificate into her keystore:

```
keytool -importcert -keystore cindy.jks -alias alice -file alice_signedby
```

The keystore verifies that the key was signed by a trusted root key that is already present in the keystore. Cindy is not asked to verify the certificate fingerprint.

Once Cindy has added the root certificate and the certificates of the people who regularly send her documents, she does not have to worry about the certicicates again until they expire.

9.3.7. Code Signing

A common use of authentication technology is signing executable programs. If you download a program, you are naturally concerned about the damage it can do. For example, the program could have been infected by a virus. If you know where the code comes from and that it has not been tampered with since it left its origin, your comfort level will be a lot higher than without this knowledge.

In this section, I'll show you how to sign JAR files, and how you can configure Java to verify the signature. This capability was designed for applets and Java Web Start applications. These are no longer commonly used technologies, but you may still need to support them in legacy products.

When Java was first released, applets ran in the “sandbox,” with limited permissions, as soon as they were loaded. If users wanted to use applets that can access the local file system, make network connections, and so on, they had to explicitly agree. To ensure that the applet code was not tampered with in transit, it had to be digitally signed.

Here is a specific example. Suppose that while surfing the Internet, you encounter a web site that offers to run an applet from an unfamiliar vendor, provided you grant it the permission to do so (see Figure 9.7). Such a program is signed with a software developer certificate issued by a certificate authority that the Java runtime trusts. The pop-up dialog box identifies the software developer and the certificate issuer. Now you need to decide whether to authorize the program.

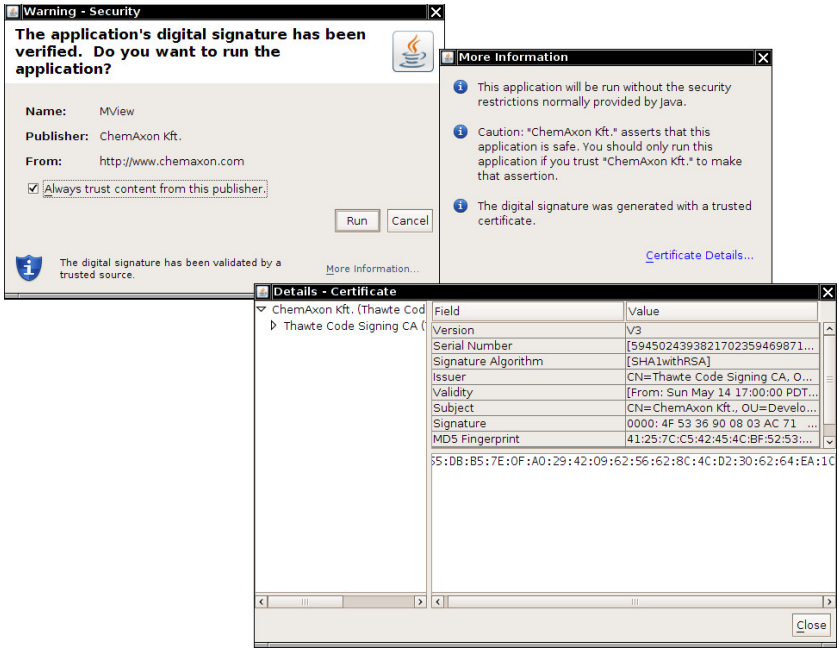


Figure 9.7: Launching a signed applet

What facts do you have at your disposal that might influence your decision? Here is what you know:

- Thawte sold a certificate to the software developer.
- The program really was signed with that certificate, and it hasn't been modified in transit.
- That certificate really was signed by Thawte—it was verified by the public key in the local cacerts file.

Of course, none of this tells you whether the code is safe to run. Can you trust a vendor if all you know is the vendor's name and the fact that Thawte sold them a software developer certificate? This approach never made much sense.

For intranet deployment, certificates are more plausible. Administrators can install policy files and certificates on local machines so that no user interaction is required for launching trusted code. However, with the deprecation of the security manager, this approach is no longer viable.

9.4. Encryption

So far, we have discussed one important cryptographic technique implemented in the Java security API—namely, authentication through digital signatures. A second important aspect of security is encryption. Even when authenticated, the information itself is plainly visible. The digital signature merely verifies that the information has not been changed. In contrast, when information is encrypted, it is not visible. It can only be decrypted with a matching key.

Authentication is sufficient for code signing—there is no need to hide the code. However, encryption is necessary when applications transfer confidential information, such as credit card numbers and other personal data.

In the past, patents and export controls prevented many companies from offering strong encryption. Fortunately, export controls are now much less stringent, and the patents for important algorithms have expired. Nowadays, Java provides excellent encryption support as a part of the standard library.

9.4.1. Symmetric Ciphers

The Java cryptographic extensions contain a class Cipher that is the superclass of all encryption algorithms. To get a cipher object, call the getInstance method:

```
Cipher cipher = Cipher.getInstance(algorithmName);
```

or

```
Cipher cipher = Cipher.getInstance(algorithmName, providerName);
```

The JDK comes with ciphers by the provider named "SunJCE". It is the default provider used if you don’t specify another provider name. You might want another provider if you need specialized algorithms that Oracle does not support.

The algorithm name is a string such as "AES" or "DES/CBC/

PKCS5Padding".

The Data Encryption Standard (DES) is a venerable block cipher with a key length of 56 bits. Nowadays, the DES algorithm is considered obsolete because it can be cracked with brute force. A far better alternative is its successor, the Advanced Encryption Standard (AES). See https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf for a detailed description of the AES algorithm. I use AES in this example.

Once you have a cipher object, initialize it by setting the mode and the key:

```
int mode = . . .;
Key key = . . .;
cipher.init(mode, key);
```

The mode is one of

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

The wrap and unwrap modes encrypt one key with another—see the next section for an example.

Now you can repeatedly call the update method to encrypt blocks of data:

```
int blockSize = cipher.getBlockSize();
var inBytes = new byte[blockSize];
. . . // read inBytes
int outputSize= cipher.getOutputSize(blockSize);
var outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0, outputSize, outBytes);
. . . // write outBytes
```

When you are done, you must call the doFinal method once. If a final block of input data is available (with fewer than blockSize bytes), call

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

If all input data have been encrypted, instead call

```
outBytes = cipher.doFinal();
```

The call to doFinal is necessary to carry out padding of the final block. Consider the DES cipher. It has a block size of eight bytes. Suppose the last block of the input data has fewer than eight bytes. Of course, we can fill the remaining bytes with 0, to obtain one final block of eight bytes, and encrypt it. But when the blocks are decrypted, the result will have several trailing 0 bytes appended to it, and therefore will be slightly different from the original input file. To avoid this problem, we need a padding scheme. A commonly used padding scheme is the one described in the Public Key Cryptography Standard (PKCS) #5 by RSA Security, Inc. (https://datatracker.ietf.org/doc/html/rfc2898).

In this scheme, the last block is not padded with a pad value of zero, but with a pad value that equals the number of pad bytes. In other words, if L is the last (incomplete) block, it is padded as follows:

```
L 01                        if length(L) = 7
L 02 02                       if length(L) = 6
L 03 03 03                      if length(L) = 5
. . .
L 07 07 07 07 07 07 07              if length(L) = 1
```

Finally, if the length of the input is actually divisible by 8, then one block

```
08 08 08 08 08 08 08 08
```

is appended to the input and encrypted. After decryption, the very last byte of the plaintext is a count of the padding characters to discard.

9.4.2. Key Generation

To encrypt, you need to generate a key. Each cipher has a different format for keys, and you need to make sure that the key generation is random. Follow these steps:

1. Get a KeyGenerator for your algorithm.
2. Initialize the generator with a source for randomness. If the block length of the cipher is variable, also specify the desired block length.
3. Call the generateKey method.

For example, here is how you generate an AES key:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
var random = new SecureRandom(); // see below
keygen.init(random);
Key key = keygen.generateKey();
```

Alternatively, you can produce a key from a fixed set of raw data (perhaps derived from a password or the timing of keystrokes). Construct a SecretKeySpec (which implements the SecretKey interface) like this:

```
byte[] keyData = . . .; // 16 bytes for AES
var key = new SecretKeySpec(keyData, "AES");
```

When generating keys, make sure you use truly random numbers. For example, the regular random number generator in the Random class, seeded by the current date and time, is not random enough. Suppose the computer clock is accurate to 1/10 of a second. Then there are at most 864,000 seeds per day. If an attacker knows the day a key was issued (which can often be deduced from a message date or certificate expiration date), it is an easy matter to generate all possible seeds for that day.

The SecureRandom class generates random numbers that are far more secure than those produced by the Random class. You still need to provide a seed to start the number sequence at a random spot. The best method for doing this is to obtain random input from a hardware device such as a white-noise generator. Another reasonable source for random input is to ask the user to type away aimlessly on the keyboard, with each

keystroke contributing only one or two bits to the random seed. Once you gather such random bits in an array of bytes, pass it to the setSeed method:

```
var random = new SecureRandom();
var b = new byte[20];
// fill with truly random bits
random.setSeed(b);
```

If you don't seed the random number generator, it will compute its own 20-byte seed by launching threads, putting them to sleep, and measuring the exact time when they are awakened.

> This algorithm is not known to be safe. In the past, algorithms that relied on the timing of some components of the computer, such as hard disk access time, were shown not to be completely random.

The sample program at the end of this section puts the AES cipher to work (see Listing 9.12). The crypt utility method in Listing 9.13 will be reused in other examples. To use the program, you first need to generate a secret key. Run

```
java v2ch09.aes.AESDemo -genkey secret.key
```

The secret key is saved in the file secret.key.

Now you can encrypt with the command

```
java v2ch09.aes.AESDemo -encrypt plaintextFile encryptedFile secre
```

Decrypt with the command

```
java v2ch09.aes.AESDemo -decrypt encryptedFile decryptedFile secr
```

The program is straightforward. The -genkey option produces a new secret key and serializes it in the given file. The -encrypt and -decrypt options both call into the same crypt method that calls the update and doFinal methods of the cipher. Note how the update method is called so long as the input blocks have the full length, and the doFinal method is either called with a

partial input block (which is then padded) or with no additional data (to generate one pad block).

Listing 9.12 **v2ch09/aes/AESDemo.java**

```
package v2ch09.aes;

import module java.base;

/**
 * This program tests the AES cipher. It can generate a secret key,
 * or encrypt or decrypt a file.
 */
class AESDemo {
   void main(String[] args) throws Exception {
      if (args[0].equals("-genkey")) {
         KeyGenerator keygen = KeyGenerator.getInstance("AES");
         var random = new SecureRandom();
         keygen.init(random);
         SecretKey key = keygen.generateKey();
         try (var out = new ObjectOutputStream(new FileOutputStream(args[1]))
            out.writeObject(key);
         }
      }
      else {
         int mode;
         if (args[0].equals("-encrypt"))
            mode = Cipher.ENCRYPT_MODE;
         else
            mode = Cipher.DECRYPT_MODE;

         try (var keyIn = new ObjectInputStream(new FileInputStream(args[3]));
               var in = new FileInputStream(args[1]);
               var out = new FileOutputStream(args[2])) {
            var key = (Key) keyIn.readObject();
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(mode, key);
            Util.crypt(in, out, cipher);
         }
      }
   }
}
```

Listing 9.13 **v2ch09/aes/Util.java**

```
package v2ch09.aes;

import module java.base;

public class Util {
   /**
    * Uses a cipher to transform the bytes in an input stream and sends the tran
    * to an output stream.
    * @param in the input stream
    * @param out the output stream
    * @param cipher the cipher that transforms the bytes
    */
   public static void crypt(InputStream in, OutputStream out, Cipher cipher)
         throws IOException, GeneralSecurityException {
      int blockSize = cipher.getBlockSize();
      int outputSize = cipher.getOutputSize(blockSize);
      var inBytes = new byte[blockSize];
      var outBytes = new byte[outputSize];

      int inLength = 0;
      boolean done = false;
      while (!done) {
         inLength = in.read(inBytes);
         if (inLength == blockSize) {
            int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
            out.write(outBytes, 0, outLength);
         }
         else
            done = true;
      }
      if (inLength > 0)
         outBytes = cipher.doFinal(inBytes, 0, inLength);
      else
         outBytes = cipher.doFinal();
      out.write(outBytes);
   }
}
```

javax.crypto.Cipher1.4

- static Cipher getInstance(String algorithmName)

- static Cipher getInstance(String algorithmName, String providerName)
  return a Cipher object that implements the specified algorithm. Throw a NoSuchAlgorithmException if the algorithm is not provided.
- int getBlockSize()
  returns the size (in bytes) of a cipher block, or 0 if the cipher is not a block cipher.
- int getOutputSize(int inputLength)
  returns the size of an output buffer that is needed if the next input has the given number of bytes. This method takes into account any buffered bytes in the cipher object.
- void init(int mode, Key key)
  initializes the cipher algorithm object. The mode is one of ENCRYPT_MODE, DECRYPT_MODE, WRAP_MODE, or UNWRAP_MODE.
- byte[] update(byte[] in)
- byte[] update(byte[] in, int offset, int length)
- int update(byte[] in, int offset, int length, byte[] out)
  transform one block of input data. The first two methods return the output. The third method returns the number of bytes placed into out.
- byte[] doFinal()
- byte[] doFinal(byte[] in)
- byte[] doFinal(byte[] in, int offset, int length)
- int doFinal(byte[] in, int offset, int length, byte[] out)
  transform the last block of input data and flush the buffer of this algorithm object. The first three methods return the output. The fourth method returns the number of bytes placed into out.

javax.crypto.KeyGenerator1.4

- static KeyGenerator getInstance(String algorithmName)
  returns a KeyGenerator object that implements the specified algorithm. Throws a NoSuchAlgorithmException if the algorithm is not provided.
- void init(SecureRandom random)
- void init(int keySize, SecureRandom random)

initialize the key generator.
- SecretKey generateKey()
  generates a new key.

javax.crypto.spec.SecretKeySpec1.4

- SecretKeySpec(byte[] key, String algorithmName)
  constructs a key specification.

9.4.3. Cipher Streams

The JCE library provides a convenient set of stream classes that automatically encrypt or decrypt stream data. For example, here is how you can encrypt data to a file:

```
Cipher cipher = . . .;
cipher.init(Cipher.ENCRYPT_MODE, key);
var out = new CipherOutputStream(new FileOutputStream(outputFi
var bytes = new byte[BLOCKSIZE];
int inLength = getData(bytes); // get data from data source
while (inLength != -1) {
   out.write(bytes, 0, inLength);
   inLength = getData(bytes); // get more data from data source
}
out.flush();
```

Similarly, you can use a CipherInputStream to read and decrypt data from a file:

```
Cipher cipher = . . .;
cipher.init(Cipher.DECRYPT_MODE, key);
var in = new CipherInputStream(new FileInputStream(inputFileNam
var bytes = new byte[BLOCKSIZE];
int inLength = in.read(bytes);
while (inLength != -1) {
   putData(bytes, inLength); // put data to destination
   inLength = in.read(bytes);
```

```
}
```

The cipher stream classes transparently handle the calls to update and doFinal, which is clearly a convenience.

javax.crypto.CipherInputStream1.4

- CipherInputStream(InputStream in, Cipher cipher)
  constructs an input stream that reads data from in and decrypts or encrypts them by using the given cipher.
- int read()
- int read(byte[] b, int off, int len)
  read data from the input stream, which is automatically decrypted or encrypted.

javax.crypto.CipherOutputStream1.4

- CipherOutputStream(OutputStream out, Cipher cipher)
  constructs an output stream that writes data to out and encrypts or decrypts them using the given cipher.
- void write(int ch)
- void write(byte[] b, int off, int len)
  write data to the output stream, which is automatically encrypted or decrypted.
- void flush()
  flushes the cipher buffer and carries out padding if necessary.

9.4.4. Public Key Ciphers

The AES cipher that you have seen in the preceding section is a symmetric cipher. The same key is used for both encryption and decryption. The Achilles heel of symmetric ciphers is key distribution. If Alice sends Bob an encrypted message, Bob needs the same key that Alice used. If Alice changes the key, she needs to send Bob both the message and, through a secure

channel, the new key. But perhaps she has no secure channel to Bob—which is why she encrypts her messages to him in the first place.

Public key cryptography solves that problem. In a public key cipher, Bob has a key pair consisting of a public key and a matching private key. Bob can publish the public key anywhere, but he must closely guard the private key. Alice simply uses the public key to encrypt her messages to Bob.

Actually, it's not quite that simple. All known public key algorithms are much slower than symmetric key algorithms such as DES or AES. It would not be practical to use a public key algorithm to encrypt large amounts of information. However, that problem can easily be overcome by combining a public key cipher with a fast symmetric cipher, like this:

1. Alice generates a random symmetric encryption key. She uses it to encrypt her plaintext.
2. Alice encrypts the symmetric key with Bob's public key.
3. Alice sends Bob both the encrypted symmetric key and the encrypted plaintext.
4. Bob uses his private key to decrypt the symmetric key.
5. Bob uses the decrypted symmetric key to decrypt the message.

Nobody but Bob can decrypt the symmetric key because only Bob has the private key for decryption. Thus, the expensive public key encryption is only applied to a small amount of key data.

The most commonly used public key algorithm is the RSA algorithm invented by Rivest, Shamir, and Adleman. Until October 2000, the algorithm was protected by a patent assigned to RSA Security, Inc. Licenses were not cheap—typically a 3% royalty, with a minimum payment of $50,000 per year. Now the algorithm is in the public domain.

To use the RSA algorithm, you need a public/private key pair. Use a KeyPairGenerator like this:

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
var random = new SecureRandom();
pairgen.initialize(KEYSIZE, random);
KeyPair keyPair = pairgen.generateKeyPair();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

The program in Listing 9.14 has three options. The -genkey option produces a key pair. The -encrypt option generates an AES key and wraps it with the public key.

```
Key key = . . .; // an AES key
Key publicKey = . . .; // a public RSA key
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

It then produces a file that contains

- The length of the wrapped key
- The wrapped key bytes
- The plaintext encrypted with the AES key

The -decrypt option decrypts such a file. To try the program, first generate the RSA keys:

```
java v2ch09.rsa.RSADemo -genkey public.key private.key
```

Then encrypt a file:

```
java v2ch09.rsa.RSADemo -encrypt plaintextFile encryptedFile publi
```

Finally, decrypt it and verify that the decrypted file matches the plaintext:

```
java v2ch09.rsa.RSADemo -decrypt encryptedFile decryptedFile priv
```

Listing 9.14 **v2ch09/rsa/RSADemo.java**

```
package v2ch09.rsa;
```

```
import module java.base;

import v2ch09.aes.Util;

/**
 * This program tests the RSA cipher. It can generate public and private key pa
 * or encrypt or decrypt a file.
 */
class RSADemo {
   final int KEYSIZE = 512;

   void main(String[] args) throws Exception {
      if (args[0].equals("-genkey")) {
         KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
         var random = new SecureRandom();
         pairgen.initialize(KEYSIZE, random);
         KeyPair keyPair = pairgen.generateKeyPair();
         try (var out = new ObjectOutputStream(new FileOutputStream(args[1]))
            out.writeObject(keyPair.getPublic());
         }
         try (var out = new ObjectOutputStream(new FileOutputStream(args[2]))
            out.writeObject(keyPair.getPrivate());
         }
      }
      else if (args[0].equals("-encrypt")) {
         KeyGenerator keygen = KeyGenerator.getInstance("AES");
         var random = new SecureRandom();
         keygen.init(random);
         SecretKey key = keygen.generateKey();

         // wrap with RSA public key
         try (var keyIn = new ObjectInputStream(new FileInputStream(args[3]));
               var out = new DataOutputStream(new FileOutputStream(args[2]));
               var in = new FileInputStream(args[1])) {
            var publicKey = (Key) keyIn.readObject();
            Cipher cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.WRAP_MODE, publicKey);
            byte[] wrappedKey = cipher.wrap(key);
            out.writeInt(wrappedKey.length);
            out.write(wrappedKey);

            cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.ENCRYPT_MODE, key);
```

```
            Util.crypt(in, out, cipher);
         }
      }
      else {
         try (var in = new DataInputStream(new FileInputStream(args[1]));
               var keyIn = new ObjectInputStream(new FileInputStream(args[3]));
               var out = new FileOutputStream(args[2])) {
            int length = in.readInt();
            var wrappedKey = new byte[length];
            in.read(wrappedKey, 0, length);

            // unwrap with RSA private key
            var privateKey = (Key) keyIn.readObject();

            Cipher cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.UNWRAP_MODE, privateKey);
            Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);

            cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.DECRYPT_MODE, key);

            Util.crypt(in, out, cipher);
         }
      }
   }
}
```

In this chapter, you had a glimpse at class loaders, and you saw the services for authentication and encryption that the Java library provides. Along the way, you perhaps shed a tear that the classic Java security architecture, based on a security manager and code signing, is now a distant memory.

In the next chapter, we will delve into advanced Swing and graphics programming.

Chapter 10 ▪ Graphical User Interface Programming

Java was born at a time when most computer users interacted with desktop applications via a graphical user interface (GUI). Nowadays, browser-based and mobile applications are far more common, but there are still times when it is useful to provide a desktop application. Moreover, many teachers and students enjoy learning Java through GUI applications. In this and the following chapter, I discuss the basics of user interface programming with the Swing toolkit. Chapter 12 covers more advanced techniques that can also be useful to generate images on a server. If you are not interested in writing GUI and graphics programs, you can safely skip these chapters.

10.1. A History of Java User Interface Toolkits

When Java 1.0 was introduced, it contained a class library, called the Abstract Window Toolkit (AWT), for basic GUI programming. The basic AWT library deals with user interface elements by delegating their creation and behavior to the

native GUI toolkit on each target platform (Windows, Linux, Macintosh, and so on). For example, if you used the original AWT to put a text box on a Java window, an underlying "peer" text box actually handled the text input. The resulting program could then, in theory, run on any of these platforms, with the "look-and-feel" of the target platform.

The peer-based approach worked well for simple applications, but it soon became apparent that it was fiendishly difficult to write a high-quality portable graphics library depending on native user interface elements. User interface elements such as menus, scrollbars, and text fields can have subtle differences in behavior on different platforms. It was hard, therefore, to give users a consistent and predictable experience with this approach. Moreover, some graphical environments (such as X11/Motif) do not have as rich a collection of user interface components as does Windows or the Macintosh. This further limits a portable library based on a "lowest common denominator" approach. As a result, GUI applications built with the AWT simply did not look as nice as native Windows or Macintosh applications, nor did they have the kind of functionality that users of those platforms had come to expect. More depressingly, there were different bugs in the AWT user interface library on the different platforms. Developers complained that they had to test their applications on each platform—a practice derisively called "write once, debug everywhere."

In 1996, Netscape created a GUI library they called the IFC (Internet Foundation Classes) that used an entirely different approach. User interface elements, such as buttons, menus, and so on, were painted onto blank windows. The only functionality required from the underlying windowing system was a way to put up a window and to paint on it. Thus, Netscape's IFC widgets looked and behaved the same no matter which platform the program ran on. Sun Microsystems worked with Netscape to perfect this approach, creating a user interface library with the code name "Swing." Swing was available as an extension to Java 1.1 and became a part of the standard library in Java 1.2.

Swing is now the official name for the non-peer-based GUI toolkit.

> Swing is not a complete replacement for the AWT—it is built on top of the AWT architecture. Swing simply gives you more capable user interface components. Whenever you write a Swing program, you use the foundations of the AWT—in particular, event handling. From now on, I will say “Swing” when I mean the “painted” user interface classes, and “AWT” when I refer to the underlying mechanisms of the windowing toolkit, such as event handling.

Swing has to work hard painting every pixel of the user interface. When Swing was first released, users complained that it was slow. (You can still get a feel for the problem if you run Swing applications on hardware such as a Raspberry Pi.) After a while, desktop computers got faster, and users complained that Swing was ugly—indeed, it had fallen behind the native widgets that had been spruced up with animations and fancy effects. More ominously, Adobe Flash was increasingly used to create user interfaces with even flashier effects that didn't use the native controls at all.

In 2007, Sun Microsystems introduced an entirely different user interface toolkit, called JavaFX, as a competitor to Flash. It ran on the Java VM but had its own programming language, called JavaFX Script. The language was optimized for programming animations and fancy effects. Programmers complained about the need to learn a new language, and they stayed away in droves. In 2011, Oracle released a new version, JavaFX 2.0, that had a Java API and no longer needed a separate programming language. Starting with Java 7 update 6, JavaFX was bundled with the JDK and JRE. However, this bundling was discontinued starting with Java 11.

Since this is a book about the core Java language and APIs, I will focus on Swing for user interface programming.

10.2. Displaying Frames

A top-level window (that is, a window that is not contained inside another window) is called a frame in Java. The AWT library has a class, called Frame, for this top level. The Swing version of this class is called JFrame and extends the Frame class. The JFrame is one of the few Swing components that is not painted on a canvas. Thus, the decorations (buttons, title bar, icons, and so on) are drawn by the user's windowing system, not by Swing.

> Most Swing component classes start with a "J": JButton, JFrame, and so on. There are classes such as Button and Frame, but they are AWT components. If you accidentally omit a "J", your program may still compile and run, but the mixture of Swing and AWT components can lead to visual and behavioral inconsistencies.

10.2.1. Creating a Frame

In this section, we will go over the most common methods for working with a Swing JFrame. Listing 10.1 lists a simple program that displays an empty frame on the screen, as illustrated in Figure 10.1.



Figure 10.1: The simplest visible frame

Listing 10.1 **v2ch10/simpleFrame/SimpleFrameDemo.java**

```
package v2ch10.simpleFrame;

import module java.desktop;

class SimpleFrameDemo {
   void main() {
      EventQueue.invokeLater(() -> {
         var frame = new SimpleFrame();
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

class SimpleFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public SimpleFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
   }
}
```

Let's work through this program, line by line.

The Swing classes are placed in the javax.swing package. The package name javax indicates a Java extension package, not a core package. For historical reasons, Swing is considered an extension. However, it is present in every Java implementation since version 1.2.

By default, a frame has a rather useless size of 0×0 pixels. We define a subclass SimpleFrame whose constructor sets the size to 300×200 pixels. This is the only difference between a SimpleFrame and a JFrame.

In the main method of the SimpleFrameDemo class, we construct a SimpleFrame object and make it visible.

There are two technical issues that we need to address in every Swing program.

First, all Swing components must be configured from the event dispatch thread, the thread of control that passes events such as mouse clicks and keystrokes to the user interface components. The following code fragment is used to execute statements in the event dispatch thread:

```
EventQueue.invokeLater(() -> {
   statements
});
```

> You will see many Swing programs that do not initialize the user interface in the event dispatch thread. It used to be perfectly acceptable to carry out the initialization in the main thread. Sadly, as Swing components got more complex, the developers of the JDK were no longer able to guarantee the safety of that approach. The probability of an error is extremely low, but you would not want to be one of the unlucky few who encounter an intermittent

---

problem. It is better to do the right thing, even if the code looks rather mysterious.

Next, we define what should happen when the user closes the application's frame. For this particular program, we want the program to exit. To select this behavior, we use the statement

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

In other programs with multiple frames, you would not want the program to exit just because the user closed one of the frames. By default, a frame is hidden when the user closes it, but the program does not terminate. (It might have been nice if the program terminated once the last frame becomes invisible, but that is not how Swing works.)

Simply constructing a frame does not automatically display it. Frames start their life invisible. That gives the programmer the chance to add components into the frame before showing it for the first time. To show the frame, the main method calls the setVisible method of the frame.

After scheduling the initialization statements, the main method exits. Note that exiting main does not terminate the program—just the main thread. The event dispatch thread keeps the program alive until it is terminated, either by closing the frame or by calling the System.exit method.

The running program is shown in Figure 10.1—it is a truly boring top-level window. As you can see in the figure, the title bar and the surrounding decorations, such as resize corners, are drawn by the operating system and not the Swing library. The Swing library draws everything inside the frame. In this program, it just fills the frame with a default background color.

10.2.2. Frame Properties

The JFrame class itself has only a few methods for changing how frames look. Of course, through the magic of inheritance, most of the methods for working with the size and position of a frame come from the various superclasses of JFrame. Here are

some of the most important methods:

- The setLocation and setBounds methods for setting the position of the frame
- The setIconImage method, which tells the windowing system which icon to display in the title bar, task switcher window, and so on
- The setTitle method for changing the text in the title bar
- The setResizable method, which takes a boolean to determine if a frame will be resizeable by the user

Figure 10.2 illustrates the inheritance hierarchy for the JFrame class.

.

Figure 10.2: Inheritance hierarchy for the frame and component classes in AWT and Swing

As the API notes indicate, the Component class (which is the ancestor of all GUI objects) and the Window class (which is the superclass of the Frame class) are where you need to look for the methods to resize and reshape frames. For example, the setLocation method in the Component class is one way to reposition a component. If you make the call

```
setLocation(x, y)
```

the top left corner is located x pixels across and y pixels down, where (0, 0) is the top left corner of the screen. Similarly, the setBounds method in Component lets you resize and relocate a component (in particular, a JFrame) in one step, as

```
setBounds(x, y, width, height)
```

Many methods of component classes come in getter/setter pairs, such as the following methods of the Frame class:

```
public String getTitle()
public void setTitle(String title)
```

Such a getter/setter pair is called a property. A property has a name and a type. The name is obtained by changing the first letter after the get or set to lowercase. For example, the Frame class has a property with name title and type String.

Conceptually, title is a property of the frame. When we set the property, we expect the title to change on the user's screen. When we get the property, we expect to get back the value that we have set.

There is one exception to the get/set convention: For properties of type boolean, the getter starts with is. For example, the following two methods define the resizable property:

```
public boolean isResizable()
public void setResizable(boolean resizable)
```

To determine an appropriate size for a frame, first find out the screen size. Call the static getDefaultToolkit method of the Toolkit class to get the Toolkit object. (The Toolkit class is a dumping ground for a variety of methods interfacing with the native windowing system.) Then call the getScreenSize method, which returns the screen size as a Dimension object. A Dimension object simultaneously stores a width and a height, in public (!) instance fields width and height. Then you can use a suitable percentage of the screen size to size the frame. Here is the code:

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
setSize(screenWidth / 2, screenHeight / 2);
```

You can also supply a frame icon:

```
Image img = new ImageIcon("icon.gif").getImage();
setIconImage(img);
```

java.awt.Component1.0

- boolean isVisible()
- void setVisible(boolean b)
  get or set the visible property. Components are initially visible, with the exception of top-level components such as JFrame.
- void setSize(int width, int height)1.1
  resizes the component to the specified width and height.
- void setLocation(int x, int y)1.1
  moves the component to a new location. The x and y coordinates use the coordinates of the container if the component is not a top-level component, or the coordinates of the screen if the component is top level (for example, a JFrame).
- void setBounds(int x, int y, int width, int height)1.1
  moves and resizes this component.
- Dimension getSize()1.1
- void setSize(Dimension d)1.1
  get or set the size property of this component.

java.awt.Frame1.0

- boolean isResizable()
- void setResizable(boolean b)
  get or set the resizable property. When the property is set, the user can resize the frame.
- String getTitle()
- void setTitle(String s)
  get or set the title property that determines the text in the title bar for the frame.
- Image getIconImage()
- void setIconImage(Image image)
  get or set the iconImage property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

java.awt.Toolkit1.0

- static Toolkit getDefaultToolkit()
  returns the default toolkit for this platform.
- Dimension getScreenSize()
  gets the size of the user's screen.

javax.swing.ImageIcon1.2

- ImageIcon(String filename)
  constructs an icon whose image is stored in a file.
- Image getImage()
  gets the image of this icon.

10.3. Displaying Information in a Component

In this section, I will show you how to display information inside a frame (Figure 10.3).



Figure 10.3: A frame that displays information

You could draw the message string directly onto a frame, but that is not considered good programming practice. In Java, frames are really designed to be containers for components, such as a menu bar and other user interface elements. You normally draw on another component which you add to the frame.

The structure of a JFrame is surprisingly complex. Look at Figure 10.4 which shows the makeup of a JFrame. As you can see, four panes are layered in a JFrame. The root pane, layered pane, and glass pane are of no interest to us; they are required to organize the menu bar and content pane and to implement the look-and-feel. The part that most concerns Swing programmers is the content pane. Any components that you add to a frame are automatically placed into the content pane:

```
Component c = . . .;
frame.add(c); // added to the content pane
```

.

Figure 10.4: Internal structure of a JFrame

In our case, we want to add a single component to the frame onto which we will draw our message. To draw on a component, you define a class that extends JComponent and override the paintComponent method in that class.

The paintComponent method has one parameter of type Graphics. A Graphics object remembers a collection of settings for drawing images and text, such as the font you set or the current color. All drawing in Java must go through a Graphics object. It has methods that draw patterns, images, and text.

Here's how to make a component onto which you can draw:

```
class MyComponent extends JComponent {
   public void paintComponent(Graphics g) {
      code for drawing
   }
```

```
}
```

Each time a window needs to be redrawn, no matter what the reason, the event handler notifies the component. This causes the paintComponent methods of all components to be executed.

Never call the paintComponent method yourself. It is called automatically whenever a part of your application needs to be redrawn, and you should not interfere with this automatic process.

What sorts of actions trigger this automatic response? For example, painting occurs when the user increases the size of the window, or minimizes and then restores the window. If the user popped up another window that covered an existing window and then made the overlaid window disappear, the window that was covered is now corrupted and will need to be repainted. (The graphics system does not save the pixels underneath.) And, of course, when the window is displayed for the first time, it needs to process the code that specifies how and where it should draw the initial elements.

> If you need to force repainting of the screen, call the repaint method instead of paintComponent. The repaint method will cause paintComponent to be called for all components, with a properly configured Graphics object.

As you saw in the code fragment above, the paintComponent method has a single parameter of type Graphics. Measurement on a Graphics object for screen display is done in pixels. The (0, 0) coordinate denotes the top left corner of the component on whose surface you are drawing.

The Graphics class has various drawing methods, and displaying text is considered a special kind of drawing. Our paintComponent method looks like this:

```
public class NotHelloWorldComponent extends JComponent {
   public static final int MESSAGE_X = 75;
   public static final int MESSAGE_Y = 100;
```

```
    public void paintComponent(Graphics g) {
       g.drawString("Not a Hello, World program", MESSAGE_X, MESSA
    }
    . . .
}
```

Finally, a component should tell its users how big it would like to be. Override the getPreferredSize method and return an object of the Dimension class with the preferred width and height:

```
public class NotHelloWorldComponent extends JComponent {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;
   . . .
   public Dimension getPreferredSize() {
      return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }
}
```

When you fill a frame with one or more components, and you simply want to use their preferred size, call the pack method instead of the setSize method:

```
class NotHelloWorldFrame extends JFrame {
   public NotHelloWorldFrame() {
      add(new NotHelloWorldComponent());
      pack();
   }
}
```

Listing 10.2 shows the complete code.

Listing 10.2 **v2ch10/notHelloWorld/NotHelloWorld.java**

```
package v2ch10.notHelloWorld;

import module java.desktop;
```

```
class NotHelloWorld {
   void main() {
      EventQueue.invokeLater(() -> {
         var frame = new NotHelloWorldFrame();
         frame.setTitle("NotHelloWorld");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * A frame that contains a message panel.
 */
class NotHelloWorldFrame extends JFrame {
   public NotHelloWorldFrame() {
      add(new NotHelloWorldComponent());
      pack();
   }
}

/**
 * A component that displays a message.
 */
class NotHelloWorldComponent extends JComponent {
   public static final int MESSAGE_X = 75;
   public static final int MESSAGE_Y = 100;

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public void paintComponent(Graphics g) {
      g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
   }

   public Dimension getPreferredSize() {
      return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }
}
```

javax.swing.JFrame 1.2

- Component add(Component c)
  adds and returns the given component to the content pane

of this frame.

java.awt.Component1.0

- void repaint()
  causes a repaint of the component “as soon as possible.”
- Dimension getPreferredSize()
  is the method to override to return the preferred size of this component.

javax.swing.JComponent1.2

- void paintComponent(Graphics g)
  is the method to override to describe how your component needs to be painted.

java.awt.Window1.0

- void pack()
  resizes this window, taking into account the preferred sizes of its components.

10.3.1. Working with 2D Shapes

Starting with Java 1.0, the Graphics class has methods to draw lines, rectangles, ellipses, and so on. But those drawing operations are very limited. We will instead use the shape classes from the Java 2D library.

To use this library, you need to obtain an object of the Graphics2D class. This class is a subclass of the Graphics class. Ever since Java 1.2, methods such as paintComponent automatically receive an object of the Graphics2D class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g) {
   Graphics2D g2 = (Graphics2D) g;
   . . .
}
```

The Java 2D library organizes geometric shapes in an object-oriented fashion. In particular, there are classes to represent lines, rectangles, and ellipses:

```
Line2D
Rectangle2D
Ellipse2D
```

These classes all implement the Shape interface. The Java 2D library supports more complex shapes—arcs, quadratic and cubic curves, and general paths—that we do not discuss in this chapter.

To draw a shape, you first create an object of a class that implements the Shape interface and then call the draw method of the Graphics2D class. For example:

```
Rectangle2D rect = . . .;
g2.draw(rect);
```

The Java 2D library uses floating-point coordinates, not integers, for pixels. Internal calculations are carried out with single-precision float quantities. Single precision is sufficient—after all, the ultimate purpose of the geometric computations is to set pixels on the screen or printer. As long as any roundoff errors stay within one pixel, the visual outcome is not affected.

However, manipulating float values is sometimes inconvenient for the programmer because Java is adamant about requiring casts when converting double values into float values. For example, consider the following statement:

```
float f = 1.2; // ERROR--possible loss of precision
```

This statement does not compile because the constant 1.2 has type double, and the compiler is nervous about loss of

precision. The remedy is to add an F suffix to the floating-point constant:

```
float f = 1.2F; // OK
```

Now consider this statement:

```
float f = r.getWidth(); // ERROR
```

This statement does not compile either, for the same reason. The getWidth method returns a double. This time, the remedy is to provide a cast:

```
float f = (float) r.getWidth(); // OK
```

These suffixes and casts are a bit of a pain, so the designers of the 2D library decided to supply two versions of each shape class: one with float coordinates for frugal programmers, and one with double coordinates for the lazy ones. (In this book, we fall into the second camp and use double coordinates whenever we can.)

The library designers chose a curious mechanism for packaging these choices. Consider the Rectangle2D class. This is an abstract class with two concrete subclasses, which are also static nested classes:

```
Rectangle2D.Float
Rectangle2D.Double
```

Figure 10.5 shows the inheritance diagram.

.

Figure 10.5: 2D rectangle classes

It is best to ignore the fact that the two concrete classes are static inner classes—that is just a gimmick to avoid names such as FloatRectangle2D and DoubleRectangle2D.

When you construct a Rectangle2D.Float object, you supply the

coordinates as float numbers. For a Rectangle2D.Double object, you supply them as double numbers.

```
var floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
var doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

The construction arguments denote the top left corner, width, and height of the rectangle.

The Rectangle2D methods have double parameter and return types. For example, the getWidth method returns a double value, even if the width is stored as a float in a Rectangle2D.Float object.

> Simply use the Double shape classes to avoid dealing with float values altogether. However, if you are constructing thousands of shape objects, consider using the Float classes to conserve memory.

What we just discussed for the Rectangle2D classes holds for the other shape classes as well. Furthermore, there is a Point2D class with subclasses Point2D.Float and Point2D.Double. Here is how to make a point object:

```
var p = new Point2D.Double(10, 20);
```

The classes Rectangle2D and Ellipse2D both inherit from the common superclass RectangularShape. Admittedly, ellipses are not rectangular, but they have a bounding rectangle (see Figure 10.6).

.

Figure 10.6: The bounding rectangle of an ellipse

The RectangularShape class defines over 20 methods that are common to these shapes, among them such useful methods as getWidth, getHeight, getCenterX, and getCenterY (but, sadly, at the time of this writing, not a getCenter method that would

return the center as a Point2D object).

Finally, a couple of legacy classes from Java 1.0 have been fitted into the shape class hierarchy. The Rectangle and Point classes, which store a rectangle and a point with integer coordinates, extend the Rectangle2D and Point2D classes.

Figure 10.7 shows the relationships between the shape classes. However, the Double and Float subclasses are omitted. Legacy classes are marked with a gray fill.

.

Figure 10.7: Relationships between the shape classes

Rectangle2D and Ellipse2D objects are simple to construct. You need to specify

- The x and y coordinates of the top left corner
- The width and height

For ellipses, these refer to the bounding rectangle. For example,

```
var e = new Ellipse2D.Double(150, 200, 100, 50);
```

constructs an ellipse that is bounded by a rectangle with the top left corner at (150, 200), width of 100, and height of 50.

When constructing an ellipse, you usually know the center, width, and height, but not the corner points of the bounding rectangle (which don't even lie on the ellipse). The setFrameFromCenter method uses the center point, but it still requires one of the four corner points. Thus, you will usually end up constructing an ellipse as follows:

```
var ellipse =
   new Ellipse2D.Double(centerX - width / 2, centerY - height / 2, widt
```

To construct a line, you supply the start and end points, either as Point2D objects or as pairs of numbers:

```
var line = new Line2D.Double(start, end);
```

or

```
var line = new Line2D.Double(startX, startY, endX, endY);
```

The program in Listing 10.3 draws a rectangle, the ellipse that is enclosed in the rectangle, a diagonal of the rectangle, and a circle that has the same center as the rectangle. Figure 10.8 shows the result.

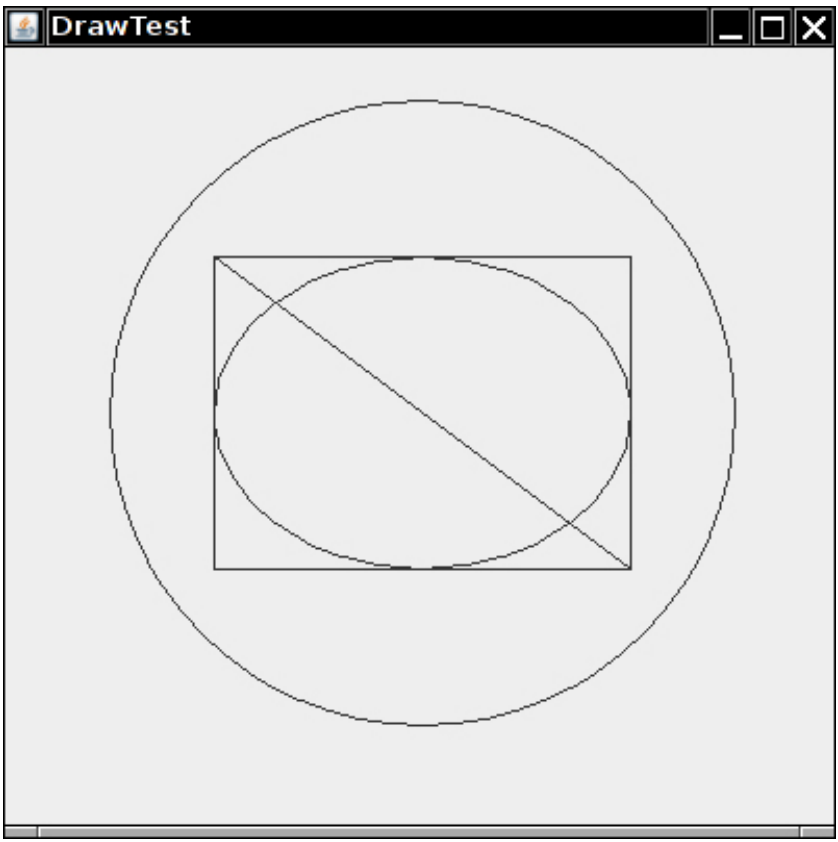


Figure 10.8: Drawing geometric shapes

Listing 10.3 **v2ch10/draw/DrawDemo.java**

```
package v2ch10.draw;

import module java.desktop;

class DrawDemo {
   void main() {
      EventQueue.invokeLater(() -> {
         var frame = new DrawFrame();
         frame.setTitle("DrawTest");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * A frame that contains a panel with drawings.
 */
class DrawFrame extends JFrame {
   public DrawFrame() {
      add(new DrawComponent());
      pack();
   }
}

/**
 * A component that displays rectangles and ellipses.
 */
class DrawComponent extends JComponent {
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;

   public void paintComponent(Graphics g) {
      var g2 = (Graphics2D) g;

      // draw a rectangle

      double leftX = 100;
      double topY = 100;
      double width = 200;
      double height = 150;

      var rect = new Rectangle2D.Double(leftX, topY, width, height);
      g2.draw(rect);
```

```
      // draw the enclosed ellipse

      var ellipse = new Ellipse2D.Double();
      ellipse.setFrame(rect);
      g2.draw(ellipse);

      // draw a diagonal line

      g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));

      // draw a circle with the same center

      double centerX = rect.getCenterX();
      double centerY = rect.getCenterY();
      double radius = 150;

      var circle = new Ellipse2D.Double();
      circle.setFrameFromCenter(centerX, centerY, centerX + radius, centerY + r
      g2.draw(circle);
   }

   public Dimension getPreferredSize() {
      return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }
}
```

java.awt.geom.RectangularShape 1.2

- double getCenterX()
- double getCenterY()
- double getMinX()
- double getMinY()
- double getMaxX()
- double getMaxY()

  return the center, minimum, or maximum x or y value of the enclosing rectangle.
- double getWidth()
- double getHeight()

  return the width or height of the enclosing rectangle.
- double getX()
- double getY()

  return the x or y coordinate of the top left corner of the

enclosing rectangle.

java.awt.geom.Rectangle2D.Double1.2

- Rectangle2D.Double(double x, double y, double w, double h)
  constructs a rectangle with the given top left corner, width, and height.

java.awt.geom.Ellipse2D.Double1.2

- Ellipse2D.Double(double x, double y, double w, double h)
  constructs an ellipse whose bounding rectangle has the given top left corner, width, and height.

java.awt.geom.Point2D.Double1.2

- Point2D.Double(double x, double y)
  constructs a point with the given coordinates.

java.awt.geom.Line2D.Double1.2

- Line2D.Double(Point2D start, Point2D end)
- Line2D.Double(double startX, double startY, double endX, double endY)
  construct a line with the given start and end points.

10.3.2. Using Color

The setPaint method of the Graphics2D class lets you select a color that is used for all subsequent drawing operations on the graphics context. For example:

```
g2.setPaint(Color.RED);
g2.drawString("Warning!", 100, 100);
```

You can fill the interiors of closed shapes (such as rectangles or ellipses) with a color. Simply call fill instead of draw:

```
Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect); // fills rect with red
```

To draw in multiple colors, select a color, draw or fill, then select another color, and draw or fill again.

> The fill method paints one fewer pixel to the right and the bottom. For example, if you draw a new Rectangle2D.Double(0, 0, 10, 20), then the drawing includes the pixels with x = 10 and y = 20. If you fill the same rectangle, those pixels are not painted.

Define colors with the Color class. The java.awt.Color class offers predefined constants for the following 13 standard colors:

```
BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY,
MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW
```

You can specify a custom color by creating a Color object by its red, green, and blue components, each a value between 0 and 255:

```
g2.setPaint(new Color(0, 128, 128)); // a dull blue-green
g2.drawString("Welcome!", 75, 125);
```

> In addition to solid colors, you can call setPaint with instances of classes that implement the Paint interface. This enables drawing with gradients and textures.

To set the background color of a component, use the setBackground method of the Component class, an ancestor of

JComponent.

```
var component = new MyComponent();
component.setBackground(Color.PINK);
```

There is also a setForeground method. It specifies the default color that is used for drawing on the component.

java.awt.Color1.0

- Color(int r, int g, int b)
  creates a color object with the given red, green, and blue components between 0 and 255.

java.awt.Graphics2D1.2

- Paint getPaint()
- void setPaint(Paint p)
  get or set the paint property of this graphics context. The Color class implements the Paint interface. Therefore, you can use this method to set the paint attribute to a solid color.
- void fill(Shape s)
  fills the shape with the current paint.

java.awt.Component1.0

- Color getForeground()
- Color getBackground()
- void setForeground(Color c)
- void setBackground(Color c)
  get or set the foreground or background color.

10.3.3. Using Fonts

The “Not a Hello World” program at the beginning of this chapter displayed a string in the default font. Sometimes, you will want to show your text in a different font. You can specify a font by its font face name. A font face name is composed of a font family name, such as “Helvetica,” and an optional suffix such as “Bold.” For example, the font faces “Helvetica” and “Helvetica Bold” are both considered to be part of the family named “Helvetica.”

To find out which fonts are available on a particular computer, call the getAvailableFontFamilyNames method of the GraphicsEnvironment class. The method returns an array of strings containing the names of all available fonts. To obtain an instance of the GraphicsEnvironment class that describes the graphics environment of the user’s system, use the static getLocalGraphicsEnvironment method. The following program prints the names of all fonts on your system:

```
import java.awt.*;

void main() {
   String[] fontNames = GraphicsEnvironment
      .getLocalGraphicsEnvironment()
      .getAvailableFontFamilyNames();
   for (String fontName : fontNames)
      IO.println(fontName);
}
```

The AWT defines five logical font names:

```
SansSerif
Serif
Monospaced
Dialog
DialogInput
```

These names are always mapped to some fonts that actually exist on the client machine. For example, on a Windows system, SansSerif is mapped to Arial.

In addition, the Oracle JDK always includes three font families

named “Lucida Sans,” “Lucida Bright,” and “Lucida Sans Typewriter.”

To draw characters in a font, you must first create an object of the class Font. Specify the font face name, the font style, and the point size. Here is an example of how you construct a Font object:

```
var sansbold14 = new Font("SansSerif", Font.BOLD, 14);
```

The third argument is the point size. Points are commonly used in typography to indicate the size of a font. There are 72 points per inch.

You can use a logical font name in place of the font face name in the Font constructor. Specify the style (plain, bold, italic, or bold italic) by setting the second Font constructor argument to one of the following values:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC
```

Here’s the code that displays the string “Hello, World!” in the standard sans serif font on your system, using 14-point bold type:

```
var sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
var message = "Hello, World!";
g2.drawString(message, 75, 100);
```

Next, let’s center the string in its component instead of drawing it at an arbitrary position. We need to know the width and height of the string in pixels. These dimensions depend on three factors:

- The font used (in our case, sans serif, bold, 14 point);
- The string (in our case, “Hello, World!”); and
- The device on which the font is drawn (in our case, the

user's screen).

To obtain an object that represents the font characteristics of the screen device, call the getFontRenderContext method of the Graphics2D class. It returns an object of the FontRenderContext class. Simply pass that object to the getStringBounds method of the Font class:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = sansbold14.getStringBounds(message, contex
```

The getStringBounds method returns a rectangle that encloses the string.

To interpret the dimensions of that rectangle, you should know some basic typesetting terms (see Figure 10.9). The baseline is the imaginary line where, for example, the bottom of a character like 'e' rests. The ascent is the distance from the baseline to the top of an ascender, which is the upper part of a letter like 'b' or 'k', or an uppercase character. The descent is the distance from the baseline to a descender, which is the lower portion of a letter like 'p' or 'g'.

.

Figure 10.9: Typesetting terms illustrated

Leading is the space between the descent of one line and the ascent of the next line. (The term has its origin from the strips of lead that typesetters used to separate lines.) The height of a font is the distance between successive baselines, which is the same as descent + leading + ascent.

The width of the rectangle that the getStringBounds method returns is the horizontal extent of the string. The height of the rectangle is the sum of ascent, descent, and leading. The rectangle has its origin at the baseline of the string. The top y coordinate of the rectangle is negative. Thus, you can obtain string width, height, and ascent as follows:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

If you need to know the descent or leading, use the getLineMetrics method of the Font class. That method returns an object of the LineMetrics class, which has methods to obtain the descent and leading:

```
LineMetrics metrics = f.getLineMetrics(message, context);
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

When you need to compute layout dimensions outside the paintComponent method, you can't obtain the font render context from the Graphics2D object. Instead, call the getFontMetrics method of the JComponent class and then call getFontRenderContext.

```
FontRenderContext context = getFontMetrics(f).getFontRender
```

To show that the positioning is accurate, the sample program in Listing 10.4 centers the string in the frame and draws the baseline and the bounding rectangle. Figure 10.10 shows the screen display.



Figure 10.10: Drawing the baseline and string bounds

Listing 10.4 **v2ch10/font/FontDemo.java**

```
package v2ch10.font;

import module java.desktop;

class FontDemo {
   void main() {
      EventQueue.invokeLater(() -> {
         var frame = new FontFrame();
         frame.setTitle("FontTest");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * A frame with a text message component.
 */
class FontFrame extends JFrame {
   public FontFrame() {
```

```
      add(new FontComponent());
      pack();
   }
}

/**
 * A component that shows a centered message in a box.
 */
class FontComponent extends JComponent {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public void paintComponent(Graphics g) {
      var g2 = (Graphics2D) g;

      String message = "Hello, World!";

      var f = new Font("Serif", Font.BOLD, 36);
      g2.setFont(f);

      // measure the size of the message

      FontRenderContext context = g2.getFontRenderContext();
      Rectangle2D bounds = f.getStringBounds(message, context);

      // set (x,y) = top left corner of text

      double x = (getWidth() - bounds.getWidth()) / 2;
      double y = (getHeight() - bounds.getHeight()) / 2;

      // add ascent to y to reach the baseline

      double ascent = -bounds.getY();
      double baseY = y + ascent;

      // draw the message

      g2.drawString(message, (int) x, (int) baseY);

      g2.setPaint(Color.LIGHT_GRAY);

      // draw the baseline

      g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(), baseY));
```

```
      // draw the enclosing rectangle

      var rect = new Rectangle2D.Double(x, y, bounds.getWidth(), bounds.getHe
      g2.draw(rect);
   }

   public Dimension getPreferredSize() {
      return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }
}
```

java.awt.Font1.0

- Font(String name, int style, int size)
  creates a new font object. The font name is either a font face name (such as "Helvetica Bold") or a logical font name (such as "Serif", "SansSerif"). The style is one of Font.PLAIN, Font.BOLD, Font.ITALIC, or Font.BOLD + Font.ITALIC.
- String getFontName()
  gets the font face name (such as "Helvetica Bold").
- String getFamily()
  gets the font family name (such as "Helvetica").
- String getName()
  gets the logical name (such as "SansSerif") if the font was created with a logical font name; otherwise, gets the font face name.
- Rectangle2D getStringBounds(String s, FontRenderContext context)1.2
  returns a rectangle that encloses the string. The origin of the rectangle falls on the baseline. The top y coordinate of the rectangle equals the negative of the ascent. The height of the rectangle equals the sum of ascent, descent, and leading. The width equals the string width.
- LineMetrics getLineMetrics(String s, FontRenderContext context)1.2
  returns a line metrics object to determine the extent of the string.

java.awt.font.LineMetrics1.2

- float getAscent()
  gets the font ascent—the distance from the baseline to the tops of uppercase characters.
- float getDescent()
  gets the font descent—the distance from the baseline to the bottoms of descenders.
- float getLeading()
  gets the font leading—the space between the bottom of one line of text and the top of the next line.
- float getHeight()
  gets the total height of the font—the distance between the two baselines of text (descent + leading + ascent).

java.awt.Graphics2D1.2

- FontRenderContext getFontRenderContext()
  gets a font render context that specifies font characteristics in this graphics context.
- void drawString(String str, float x, float y)
  draws a string in the current font and color.

javax.swing.JComponent1.2

- FontMetrics getFontMetrics(Font f)5.0
  gets the font metrics for the given font. The FontMetrics class is a precursor to the LineMetrics class.

java.awt.FontMetrics1.0

- FontRenderContext getFontRenderContext()1.2
  gets a font render context for the font.

10.3.4. Displaying Images

You can use the ImageIcon class to read an image from a file:

```
Image image = new ImageIcon(filename).getImage();
```

Now the variable image contains a reference to an object that encapsulates the image data. Display the image with the drawImage method of the Graphics class.

```
public void paintComponent(Graphics g) {
   . . .
   g.drawImage(image, x, y, null);
}
```

We can take this a little bit further and tile the window with the graphics image. The result looks like the screen shown in Figure 10.11. We do the tiling in the paintComponent method. We first draw one copy of the image in the top left corner and then use the copyArea call to copy it into the entire window:

```
for (int i = 0; i * imageWidth <= getWidth(); i++)
   for (int j = 0; j * imageHeight <= getHeight(); j++)
      if (i + j > 0)
         g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j *
```

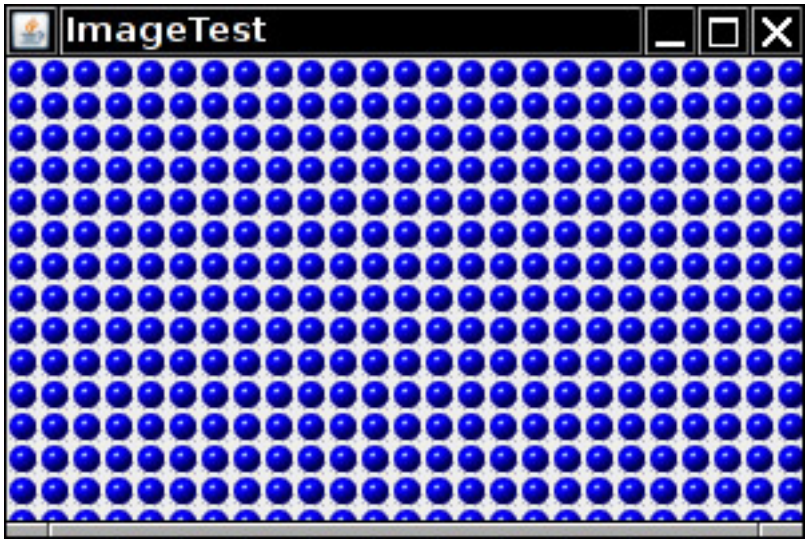


Figure 10.11: Window with tiled graphics image

Listing 10.5 **v2ch10/image/ImageDemo.java**

```
package v2ch10.image;

import module java.desktop;

class ImageDemo {
   void main() {
      EventQueue.invokeLater(() -> {
         JFrame frame = new ImageFrame();
         frame.setTitle("ImageTest");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * A frame with an image component.
 */
class ImageFrame extends JFrame {
   public ImageFrame() {
```

```
      add(new ImageComponent());
      pack();
   }
}

/**
 * A component that displays a tiled image.
 */
class ImageComponent extends JComponent {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   private Image image;

   public ImageComponent() {
      image = new ImageIcon("images/blue-ball.gif").getImage();
   }

   public void paintComponent(Graphics g) {
      if (image == null) return;

      int imageWidth = image.getWidth(null);
      int imageHeight = image.getHeight(null);
      if (imageWidth <= 0 || imageHeight <= 0) return;

      // draw the image in the upper-left corner

      g.drawImage(image, 0, 0, null);
      // tile the image across the component

      for (int i = 0; i * imageWidth <= getWidth(); i++)
         for (int j = 0; j * imageHeight <= getHeight(); j++)
            if (i + j > 0)
               g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j * imag
   }

   public Dimension getPreferredSize() {
      return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }
}
```

java.awt.Graphics1.0

■ boolean drawImage(Image img, int x, int y, ImageObserver

observer)

- boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)
  draw an unscaled or scaled image. Note: This call may return before the image is drawn. The imageObserver object is notified of the rendering progress. This was a useful feature in the distant past. Nowadays, just pass a null observer.
- void copyArea(int x, int y, int width, int height, int dx, int dy)
  copies an area of the screen. The dx and dy parameters are the distance from the source area to the target area.

10.4. Event Handling

Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks. These events are then reported to the programs that are running. Each program then decides what, if anything, to do in response to these events.

10.4.1. Basic Event Handling Concepts

In the Java AWT, event sources (such as buttons or scrollbars) have methods that allow you to register event listeners—objects that carry out the desired response to the event.

When an event listener is notified about an event, information about the event is encapsulated in an event object. In Java, all event objects ultimately derive from the class java.util.EventObject. Of course, there are subclasses for each event type, such as ActionEvent and WindowEvent.

Different event sources can produce different kinds of events. For example, a button can send ActionEvent objects, whereas a window can send WindowEvent objects.

To sum up, here's an overview of how event handling in the

AWT works:

- An event listener is an instance of a class that implements a listener interface.
- An event source is an object that can register listener objects and send them event objects.
- The event source sends out event objects to all registered listeners when that event occurs.
- The listener objects then use the information in the event object to determine their reaction to the event.

Figure 10.12 shows the relationship between the event handling classes and interfaces.

.

Figure 10.12: Relationship between event sources and listeners

Here is an example for specifying a listener:

```
ActionListener listener = . . .;
var button = new JButton("OK");
button.addActionListener(listener);
```

Now the listener object is notified whenever an “action event” occurs in the button. For buttons, as you might expect, an action event is a button click.

To implement the ActionListener interface, the listener class must have a method called actionPerformed that receives an ActionEvent object as a parameter.

```
class MyListener implements ActionListener {
   . . .
   public void actionPerformed(ActionEvent event) {
      // reaction to button click goes here
      . . .
   }
}
```

Whenever the user clicks the button, the JButton object creates an ActionEvent object and calls listener.actionPerformed(event), passing that event object. An event source such as a button can have multiple listeners. In that case, the button calls the actionPerformed methods of all listeners whenever the user clicks the button.

Figure 10.13 shows the interaction between the event source, event listener, and event object.

.

Figure 10.13: Event notification

10.4.2. Example: Handling a Button Click

As a way of getting comfortable with the event delegation model, let's work through all the details needed for the simple example of responding to a button click. For this example, we will show a panel populated with three buttons. Three listener objects are added as action listeners to the buttons.

With this scenario, each time a user clicks on any of the buttons on the panel, the associated listener object receives an ActionEvent that indicates a button click. In our sample program, the listener object will then change the background color of the panel.

Before I can show you the program that listens to button clicks, I need to explain how to create buttons and how to add them to a panel.

To create a button, specify a label string, an icon, or both in the button constructor. Here are two examples:

```
var yellowButton = new JButton("Yellow");
var blueButton = new JButton(new ImageIcon("blue-ball.gif"));
```

Call the add method to add the buttons to a panel:

```
var yellowButton = new JButton("Yellow");
var blueButton = new JButton("Blue");
var redButton = new JButton("Red");

buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);
```

Figure 10.14 shows the result.



Figure 10.14: A panel filled with buttons

Next, we need to add code that listens to these buttons. This requires classes that implement the ActionListener interface, which, as I just mentioned, has one method: actionPerformed, whose signature looks like this:

```
public void actionPerformed(ActionEvent event)
```

The way to use the ActionListener interface is the same in all situations: The actionPerformed method (which is the only method in ActionListener) has an object of type ActionEvent as a parameter. This event object gives you information about the

event that happened.

When a button is clicked, we want the background color of the panel to change to a particular color. We store the desired color in our listener class.

```
class ColorAction implements ActionListener {
   private Color backgroundColor;

   public ColorAction(Color c) {
      backgroundColor = c;
   }

   public void actionPerformed(ActionEvent event) {
      // set panel background color
      . . .
   }
}
```

We then construct one object for each color and set the objects as the button listeners.

```
var yellowAction = new ColorAction(Color.YELLOW);
var blueAction = new ColorAction(Color.BLUE);
var redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

For example, if a user clicks on the button marked “Yellow”, the actionPerformed method of the yellowAction object is called. Its backgroundColor instance field is set to Color.YELLOW, and it can now proceed to set the panel’s background color.

Just one issue remains. The ColorAction object doesn’t have access to the buttonPanel variable. You can solve this problem in two ways. You can store the panel in the ColorAction object and set it in the ColorAction constructor. Or, more conveniently, you can make ColorAction into an inner class of the ButtonFrame class. Its methods can then access the outer panel

automatically.

Listing 10.6 contains the complete frame class. Whenever you click one of the buttons, the appropriate action listener changes the background color of the panel.

Listing 10.6 **v2ch10/button/ButtonFrame.java**

```
package v2ch10.button;

import module java.desktop;

/**
 * A frame with a button panel.
 */
public class ButtonFrame extends JFrame {
   private JPanel buttonPanel;
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public ButtonFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // create buttons
      var yellowButton = new JButton("Yellow");
      var blueButton = new JButton("Blue");
      var redButton = new JButton("Red");

      buttonPanel = new JPanel();

      // add buttons to panel
      buttonPanel.add(yellowButton);
      buttonPanel.add(blueButton);
      buttonPanel.add(redButton);

      // add panel to frame
      add(buttonPanel);

      // create button actions
      var yellowAction = new ColorAction(Color.YELLOW);
      var blueAction = new ColorAction(Color.BLUE);
      var redAction = new ColorAction(Color.RED);
```

```
      // associate actions with buttons
      yellowButton.addActionListener(yellowAction);
      blueButton.addActionListener(blueAction);
      redButton.addActionListener(redAction);
   }

   /**
    * An action listener that sets the panel's background color.
    */
   private class ColorAction implements ActionListener {
      private Color backgroundColor;

      public ColorAction(Color c) {
         backgroundColor = c;
      }

      public void actionPerformed(ActionEvent event) {
         buttonPanel.setBackground(backgroundColor);
      }
   }
}
```

javax.swing.JButton1.2

- JButton(String label)
- JButton(Icon icon)
- JButton(String label, Icon icon)
  construct a button. The label string can be plain text or HTML; for example, "<html>Ok</html>".

java.awt.Container1.0

- Component add(Component c)
  adds the component c to this container.

10.4.3. Specifying Listeners Concisely

In the preceding section, we defined a class for the event

listener and constructed three objects of that class. It is not all that common to have multiple instances of a listener class. Most commonly, each listener carries out a separate action. In that case, there is no need to make a separate class. Simply use a lambda expression:

```
exitButton.addActionListener(_ -> System.exit(0));
```

> Recall that the _ denotes an unused parameter. Prior to Java 22, use a variable name such as event instead.

Now consider the case in which we have multiple related actions, such as the color buttons of the preceding section. In such a case, implement a helper method:

```
public void makeButton(String name, Color backgroundColor) {
   var button = new JButton(name);
   buttonPanel.add(button);
   button.addActionListener(_ -> buttonPanel.setBackground(backgr
}
```

Note that the lambda expression refers to the parameter variable backgroundColor.

Then we simply call

```
makeButton("yellow", Color.YELLOW);
makeButton("blue", Color.BLUE);
makeButton("red", Color.RED);
```

Here, we construct three listener objects, one for each color, without explicitly defining a class. Each time the helper method is called, it makes an instance of a class that implements the ActionListener interface. Its actionPerformed action references the backGroundColor value that is, in fact, stored with the listener object. However, all this happens without you having to explicitly define listener classes, instance fields, or constructors that set them.

In older code, you may see the use of anonymous classes:

```
exitButton.addActionListener(new ActionListener() {
   public void actionPerformed(new ActionEvent) {
      System.exit(0);
   }
});
```

Of course, this rather verbose code is no longer necessary. Using a lambda expression is simpler and clearer.

10.4.4. Adapter Classes

Not all events are as simple to handle as button clicks. Suppose you want to monitor when the user tries to close the main frame in order to put up a dialog and exit the program only when the user agrees.

When the user tries to close a window, the JFrame object is the source of a WindowEvent. If you want to catch that event, you must have an appropriate listener object and add it to the frame's list of window listeners.

```
WindowListener listener = . . .;
frame.addWindowListener(listener);
```

The window listener must be an object of a class that implements the WindowListener interface. There are actually seven methods in the WindowListener interface. The frame calls them as the responses to seven distinct events that could happen to a window. The names are self-explanatory, except that "iconified" is usually called "minimized" under Windows. Here is the complete WindowListener interface:

```
public interface WindowListener {
   void windowOpened(WindowEvent e);
   void windowClosing(WindowEvent e);
   void windowClosed(WindowEvent e);
   void windowIconified(WindowEvent e);
   void windowDeiconified(WindowEvent e);
```

```
    void windowActivated(WindowEvent e);
    void windowDeactivated(WindowEvent e);
}
```

Of course, we can define a class that implements the interface, add a call to System.exit(0) in the windowClosing method, and write do-nothing functions for the other six methods. However, typing code for six methods that don't do anything is the kind of tedious busywork that nobody likes. To simplify this task, each of the AWT listener interfaces that have more than one method comes with a companion adapter class that implements all the methods in the interface but does nothing with them. For example, the WindowAdapter class has seven do-nothing methods. You extend the adapter class to specify the desired reactions to some, but not all, of the event types in the interface. (An interface such as ActionListener that has only a single method does not need an adapter class.)

Here is how we can define a window listener that overrides the windowClosing method:

```
class Terminator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        if (user agrees)
            System.exit(0);
    }
}
```

Now you can register an object of type Terminator as the event listener:

```
var listener = new Terminator();
frame.addWindowListener(listener);
```

Nowadays, one would implement do-nothing methods of the WindowListener interface as default methods. However, Swing was invented many years before there were default methods.

---

java.awt.event.WindowListener1.1

- void windowOpened(WindowEvent e)
  is called after the window has been opened.
- void windowClosing(WindowEvent e)
  is called when the user has issued a window manager command to close the window. Note that the window will close only if its hide or dispose method is called.
- void windowClosed(WindowEvent e)
  is called after the window has closed.
- void windowIconified(WindowEvent e)
  is called after the window has been iconified.
- void windowDeiconified(WindowEvent e)
  is called after the window has been deiconified.
- void windowActivated(WindowEvent e)
  is called after the window has become active. Only a frame or dialog can be active. Typically, the window manager decorates the active window—for example, by highlighting the title bar.
- void windowDeactivated(WindowEvent e)
  is called after the window has become deactivated.

java.awt.event.WindowStateListener1.4

- void windowStateChanged(WindowEvent event)
  is called after the window has been maximized, iconified, or restored to normal size.

10.4.5. Actions

It is common to have multiple ways to activate the same command. The user can choose a certain function through a menu, a keystroke, or a button on a toolbar. This is easy to achieve in the AWT event model: link all events to the same listener. For example, suppose blueAction is an action listener whose actionPerformed method changes the background color to blue. You can attach the same object as a listener to several

event sources:

- A toolbar button labeled “Blue”
- A menu item labeled “Blue”
- A keystroke Ctrl+B

The color change command will now be handled in a uniform way, no matter whether it was caused by a button click, a menu selection, or a key press.

The Swing package provides a very useful mechanism to encapsulate commands and to attach them to multiple event sources: the Action interface. An action is an object that encapsulates

- A description of the command (as a text string and an optional icon); and
- Parameters that are necessary to carry out the command (such as the requested color in our example).

The Action interface has the following methods:

```
void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void putValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listen
```

The first method is the familiar method in the ActionListener interface; in fact, the Action interface extends the ActionListener interface. Therefore, you can use an Action object whenever an ActionListener object is expected.

The next two methods let you enable or disable the action and check whether the action is currently enabled. When an action is attached to a menu or toolbar and the action is disabled, the option is grayed out.

The putValue and getValue methods let you store and retrieve

arbitrary name/value pairs in the action object. A couple of important predefined strings, namely Action.NAME and Action.SMALL_ICON, store action names and icons into an action object:

```
action.putValue(Action.NAME, "Blue");
action.putValue(Action.SMALL_ICON, new ImageIcon("blue-ball.gif"))
```

Table 10.1 shows all predefined action table names.

Name	Value
NAME	Name of the action, displayed on buttons and menu items.
SMALL_ICON	Name of a small icon for display in a button, menu item, or toolbar.
SHORT_DESCRIPTION	A short description of the icon for display in a tooltip.
LONG_DESCRIPTION	A long description of the icon for potential use in online help. No Swing component uses this value.
MNEMONIC_KEY	An abbreviation for display in menu items.
ACCELERATOR_KEY	A keystroke for an accelerator keystroke. No Swing component uses this value.
ACTION_COMMAND_KEY	Actionally, used in the now-obsolete registerKeyboardAction method.
DEFAULT	Potentially useful catch-all property. No Swing component uses this value.

Table 10.1: Predefined Action Table Names

If the action object is added to a menu or toolbar, the name and icon are automatically retrieved and displayed in the menu item or toolbar button. The SHORT_DESCRIPTION value turns into a tooltip.

The final two methods of the Action interface allow other objects, in particular menus or toolbars that trigger the action, to be notified when the properties of the action object change. For example, if a menu is added as a property change listener of an action object and the action object is subsequently

disabled, the menu is called and can gray out the action name.

Note that Action is an interface, not a class. Any class implementing this interface must implement the seven methods we just discussed. Fortunately, a friendly soul has provided a class AbstractAction that implements all methods except for actionPerformed. That class takes care of storing all name/value pairs and managing the property change listeners. You simply extend AbstractAction and supply an actionPerformed method.

Let's build an action object that can execute color change commands. We store the name of the command, an icon, and the desired color. We store the color in the table of name/value pairs that the AbstractAction class provides. Here is the code for the ColorAction class. The constructor sets the name/value pairs, and the actionPerformed method carries out the color change action.

```
public class ColorAction extends AbstractAction {
   public ColorAction(String name, Icon icon, Color c) {
      putValue(Action.NAME, name);
      putValue(Action.SMALL_ICON, icon);
      putValue("color", c);
      putValue(Action.SHORT_DESCRIPTION, "Set panel color to " + na
   }

   public void actionPerformed(ActionEvent event) {
      Color c = (Color) getValue("color");
      buttonPanel.setBackground(c);
   }
}
```

Our test program creates three objects of this class, such as

```
var blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.
```

Next, let's associate this action with a button. That is easy because we can use a JButton constructor that takes an Action object.

```
var blueButton = new JButton(blueAction);
```

That constructor reads the name and icon from the action, sets the short description as the tooltip, and sets the action as the listener. You can see the icons and a tooltip in Figure 10.15.

As demonstrated in the next chapter, it is just as easy to add the same action to a menu.

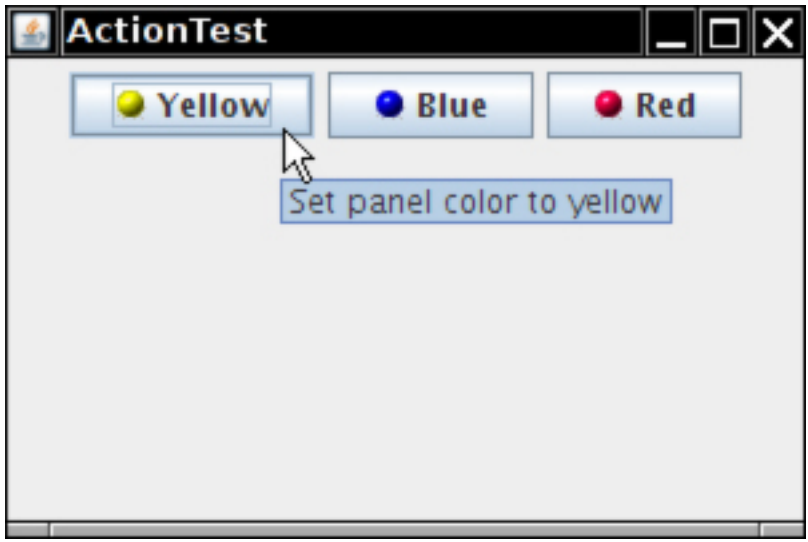


Figure 10.15: Buttons display the icons from the action objects.

Finally, we want to add the action objects to keystrokes so that an action is carried out when the user types a keyboard command. To associate actions with keystrokes, you first need to generate objects of the KeyStroke class. This convenience class encapsulates the description of a key. To generate a KeyStroke object, don't call a constructor but instead use the static getKeyStroke method of the KeyStroke class.

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl B");
```

To understand the next step, you need to understand the concept of keyboard focus. A user interface can have many

buttons, menus, scrollbars, and other components. When you hit a key, it is sent to the component that has focus. That component is usually (but not always) visually distinguished. For example, in the Java look-and-feel, a button with focus has a thin rectangular border around the button text. You can use the Tab key to move the focus between components. When you press the space bar, the button with focus is clicked. Other keys carry out different actions; for example, the arrow keys can move a scrollbar.

However, in our case, we do not want to send the keystroke to the component that has focus. Otherwise, each of the buttons would need to know how to handle the Ctrl+Y, Ctrl+B, and Ctrl +R keys.

This is a common problem, and the Swing designers came up with a convenient solution. Every JComponent has three input maps, each mapping KeyStroke objects to associated actions. The three input maps correspond to three different conditions (see Table 10.2).

Key Stroke Action	
WHEN_FOCUSED	When this component has keyboard focus
WHEN_ANCESTOR_OF_FOCUSED_COMPONENT	When it contains the component that has keyboard focus
WHEN_IN_FOCUSED_WINDOW	When it contains the component that has keyboard focus

Table 10.2: Input Map Conditions

Keystroke processing checks these maps in the following order:

1. Check the WHEN_FOCUSED map of the component with input focus. If the keystroke exists and its corresponding action is enabled, execute the action and stop processing.
2. Starting from the component with input focus, check the WHEN_ANCESTOR_OF_FOCUSED_COMPONENT maps of its parent components. As soon as a map with the keystroke and a corresponding enabled action is found, execute the action and stop processing.

3. Look at all visible and enabled components, in the window with input focus, that have this keystroke registered in a WHEN_IN_FOCUSED_WINDOW map. Give these components (in the order of their keystroke registration) a chance to execute the corresponding action. As soon as the first enabled action is executed, stop processing.

To obtain an input map from the component, use the getInputMap method. Here is an example:

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED)
```

The WHEN_FOCUSED condition means that this map is consulted when the current component has the keyboard focus. In our situation, that isn't the map we want. One of the buttons, not the panel, has the input focus. Either of the other two map choices works fine for inserting the color change keystrokes. We use WHEN_ANCESTOR_OF_FOCUSED_COMPONENT in our example program.

The InputMap doesn't directly map KeyStroke objects to Action objects. Instead, it maps to arbitrary objects, and a second map, implemented by the ActionMap class, maps objects to actions. That makes it easier to share the same actions among keystrokes that come from different input maps.

Thus, each component has three input maps and one action map. To tie them together, you need to come up with names for the actions. Here is how you can tie a key to an action:

```
imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
ActionMap amap = panel.getActionMap();
amap.put("panel.yellow", yellowAction);
```

It is customary to use the string "none" for a do-nothing action. That makes it easy to deactivate a key:

```
imap.put(KeyStroke.getKeyStroke("ctrl C"), "none");
```

The JDK documentation suggests using the action name as the action's key. I don't think that is a good idea. The action name is displayed on buttons and menu items; thus, it can change at the whim of the UI designer and may be translated into multiple languages. Such unstable strings are poor choices for lookup keys, so I recommend that you come up with action names that are independent of the displayed names.

To summarize, here is what you do to carry out the same action in response to a button, a menu item, or a keystroke:

1. Implement a class that extends the AbstractAction class. You may be able to use the same class for multiple related actions.
2. Construct an object of the action class.
3. Construct a button or menu item from the action object. The constructor will read the label text and icon from the action object.
4. For actions that can be triggered by keystrokes, you have to carry out additional steps. First, locate the top-level component of the window, such as a panel that contains all other components.
5. Then, get the WHEN_ANCESTOR_OF_FOCUSED_COMPONENT input map of the top-level component. Make a KeyStroke object for the desired keystroke. Make an action key object, such as a string that describes your action. Add the pair (keystroke, action key) into the input map.
6. Finally, get the action map of the top-level component. Add the pair (action key, action object) into the map.

javax.swing.Action 1.2

- boolean isEnabled()
- void setEnabled(boolean b)
  get or set the enabled property of this action.

- void putValue(String key, Object value)
  places a key/value pair inside the action object. The key can be any string, but several names have predefined meanings—see .
- Object getValue(String key)
  returns the value of a stored name/value pair.

javax.swing.KeyStroke1.2

- static KeyStroke getKeyStroke(String description)
  constructs a keystroke from a human-readable description (a sequence of whitespace-delimited strings). The description starts with zero or more modifiers (shift, control, ctrl, meta, alt, altGraph) and ends with either the string typed, followed by a one-character string (for example, "typed a"), or an optional event specifier (pressed or released, with pressed being the default), followed by a key code. The key code, when prefixed with VK_, should correspond to a KeyEvent constant; for example, "INSERT" corresponds to KeyEvent.VK_INSERT.

javax.swing.JComponent1.2

- ActionMap getActionMap()1.3
  returns the map that associates action map keys (which can be arbitrary objects) with Action objects.
- InputMap getInputMap(int flag)1.3
  gets the input map that maps key strokes to action map keys. The flag is one of the values in .

10.4.6. Mouse Events

You do not need to handle mouse events explicitly if you just want the user to be able to click on a button or menu. These mouse operations are handled internally by the various components in the user interface. However, if you want to

enable the user to draw with the mouse, you will need to trap the mouse move, click, and drag events.

In this section, I will show you a simple graphics editor application that allows the user to place, move, and erase squares on a canvas (see Figure 10.16).

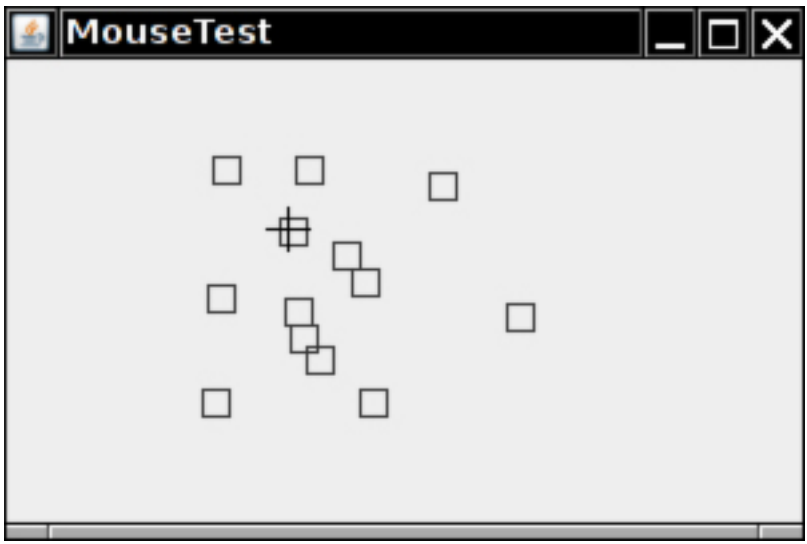


Figure 10.16: A mouse test program

When the user clicks a mouse button, three listener methods are called: mousePressed when the mouse is first pressed, mouseReleased when the mouse is released, and, finally, mouseClicked. If you are only interested in complete clicks, you can ignore the first two methods. By using the getX and getY methods on the MouseEvent parameter, you can obtain the x and y coordinates of the mouse pointer when the mouse was clicked. To distinguish between single, double, and triple (!) clicks, use the getClickCount method.

In our sample program, we supply both a mousePressed and a mouseClicked method. When you click on a pixel that is not inside any of the squares that have been drawn, a new square is added. We implemented this in the mousePressed method so

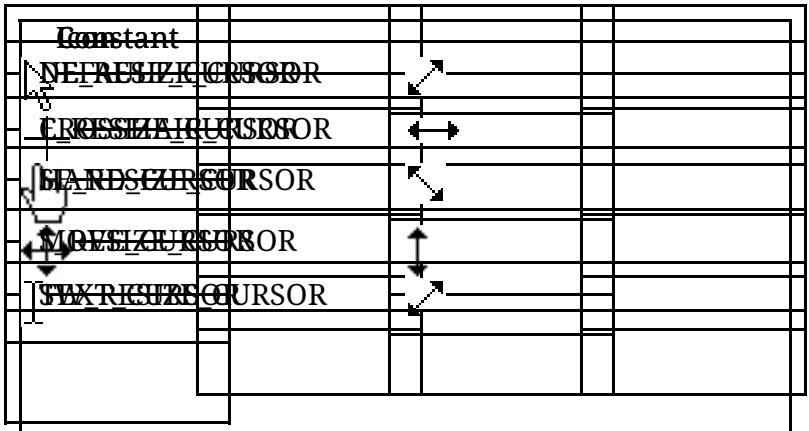
that the user receives immediate feedback and does not have to wait until the mouse button is released. When a user double-clicks inside an existing square, it is erased. We implemented this in the mouseClicked method because we need the click count.

```
public void mousePressed(MouseEvent event) {
   current = find(event.getPoint());
   if (current == null) // not inside a square
      add(event.getPoint());
}

public void mouseClicked(MouseEvent event) {
   current = find(event.getPoint());
   if (current != null && event.getClickCount() >= 2)
      remove(current);
}
```

As the mouse moves over a window, the window receives a steady stream of mouse movement events. Note that there are separate MouseListener and MouseMotionListener interfaces. This is done for efficiency—there are a lot of mouse events as the user moves the mouse around, and a listener that just cares about mouse clicks will not be bothered with unwanted mouse moves.

Our test application traps mouse motion events to change the cursor to a different shape (a cross hair) when it is over a square. This is done with the getPredefinedCursor method of the Cursor class. Table 10.3 lists the constants to use with this method along with what the cursors look like under Windows.



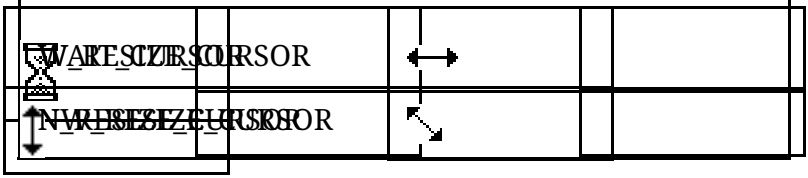


Table 10.3: Sample Cursor Shapes

Here is the mouseMoved method of the MouseMotionListener in our example program:

```
public void mouseMoved(MouseEvent event) {
   if (find(event.getPoint()) == null)
      setCursor(Cursor.getDefaultCursor());
   else
      setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURS
}
```

If the user presses a mouse button while the mouse is in motion, mouseDragged calls are generated instead of mouseMoved calls. Our test application lets a user drag the square under the cursor. We simply update the currently dragged rectangle to be centered under the mouse position. Then, we repaint the canvas to show the new mouse position.

```
public void mouseDragged(MouseEvent event) {
   if (current != null) {
      int x = event.getX();
      int y = event.getY();

      current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDE
      repaint();
   }
}
```

The mouseMoved method is only called as long as the mouse stays inside the component. However, the mouseDragged method keeps getting called even when the mouse is being dragged outside the component.

There are two other mouse event methods: mouseEntered and

mouseExited. These methods are called when the mouse enters or exits a component.

Finally, let's see how to listen to mouse events. Mouse clicks are reported through the mouseClicked method, which is part of the MouseListener interface. Many applications are only interested in mouse clicks and not in mouse moves; with the mouse move events occurring so frequently, the mouse move and drag events are defined in a separate interface called MouseMotionListener.

In our program we are interested in both types of mouse events. We define two inner classes: MouseHandler and MouseMotionHandler. The MouseHandler class extends the MouseAdapter class because it defines only two of the five MouseListener methods. (The MouseAdapter class defines all of them as do-nothing methods.) The MouseMotionHandler implements the MouseMotionListener and defines both methods of that interface. Listing 10.7 is the program listing.

Listing 10.7 **v2ch10/mouse/MouseComponent.java**

```
package v2ch10.mouse;

import module java.base;
import module java.desktop;
import java.awt.event.MouseEvent;

/**
 * A component with mouse operations for adding and removing squares.
 */
public class MouseComponent extends JComponent {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   private static final int SIDELENGTH = 10;
   private ArrayList<Rectangle2D> squares;
   private Rectangle2D current; // the square containing the mouse cursor

   public MouseComponent() {
      squares = new ArrayList<>();
```

```
      current = null;

      addMouseListener(new MouseHandler());
      addMouseMotionListener(new MouseMotionHandler());
   }

   public Dimension getPreferredSize() {
      return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }

   public void paintComponent(Graphics g) {
      var g2 = (Graphics2D) g;

      // draw all squares
      for (Rectangle2D r : squares)
         g2.draw(r);
   }

   /**
    * Finds the first square containing a point.
    * @param p a point
    * @return the first square that contains p
    */
   public Rectangle2D find(Point2D p) {
      for (Rectangle2D r : squares) {
         if (r.contains(p)) return r;
      }
      return null;
   }

   /**
    * Adds a square to the collection.
    * @param p the center of the square
    */
   public void add(Point2D p) {
      double x = p.getX();
      double y = p.getY();

      current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH /
            SIDELENGTH);
      squares.add(current);
      repaint();
   }
```

```
/**
 * Removes a square from the collection.
 * @param s the square to remove
 */
public void remove(Rectangle2D s) {
   if (s == null) return;
   if (s == current) current = null;
   squares.remove(s);
   repaint();
}

private class MouseHandler extends MouseAdapter {
   public void mousePressed(MouseEvent event) {
      // add a new square if the cursor isn't inside a square
      current = find(event.getPoint());
      if (current == null) add(event.getPoint());
   }

   public void mouseClicked(MouseEvent event) {
      // remove the current square if double clicked
      current = find(event.getPoint());
      if (current != null && event.getClickCount() >= 2) remove(current);
   }
}

private class MouseMotionHandler implements MouseMotionListener {
   public void mouseMoved(MouseEvent event) {
      // set the mouse cursor to cross hairs if it is inside a rectangle

      if (find(event.getPoint()) == null)
         setCursor(Cursor.getDefaultCursor());
      else
         setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
   }

   public void mouseDragged(MouseEvent event) {
      if (current != null) {
         int x = event.getX();
         int y = event.getY();

         // drag the current rectangle to center it at (x, y)
         current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENG
               SIDELENGTH);
         repaint();
```

```
        }
      }
    }
}
```

java.awt.event.MouseEvent1.1

- int getX()
- int getY()
- Point getPoint()
  return the x (horizontal) and y (vertical) coordinates of the point where the event happened, measured from the top left corner of the component that is the event source.
- int getClickCount()
  returns the number of consecutive mouse clicks associated with this event. (The time interval for what constitutes “consecutive” is system-dependent.)

java.awt.Component1.0

- public void setCursor(Cursor cursor)1.1
  sets the cursor image to the specified cursor.

10.4.7. The AWT Event Hierarchy

The EventObject class has a subclass AWTEvent, which is the parent of all AWT event classes. Figure 10.17 shows the inheritance diagram of the AWT events.

.

Figure 10.17: Inheritance diagram of AWT event classes

Some of the Swing components generate event objects of yet more event types; these directly extend EventObject, not AWTEvent.

The event objects encapsulate information about the event that the event source communicates to its listeners. When necessary, you can then analyze the event objects that were passed to the listener object, as we did in the button example with the getSource and getActionCommand methods.

Some of the AWT event classes are of no practical use for the Java programmer. For example, the AWT inserts PaintEvent objects into the event queue, but these objects are not delivered to listeners. Java programmers don't listen to paint events; instead, they override the paintComponent method to control repainting. The AWT also generates a number of events that are needed only by systems programmers, to provide input systems for ideographic languages, automated testing robots, and so on.

The AWT makes a useful distinction between low-level and semantic events. A semantic event is one that expresses what the user is doing, such as "clicking that button"; an ActionEvent is a semantic event. Low-level events are those events that make this possible. In the case of a button click, this is a mouse down, a series of mouse moves, and a mouse up (but only if the mouse up is inside the button area). Or it might be a keystroke, which happens if the user selects the button with the Tab key and then activates it with the space bar. Similarly, adjusting a scrollbar is a semantic event, but dragging the mouse is a low-level event.

Here are the most commonly used semantic event classes in the java.awt.event package:

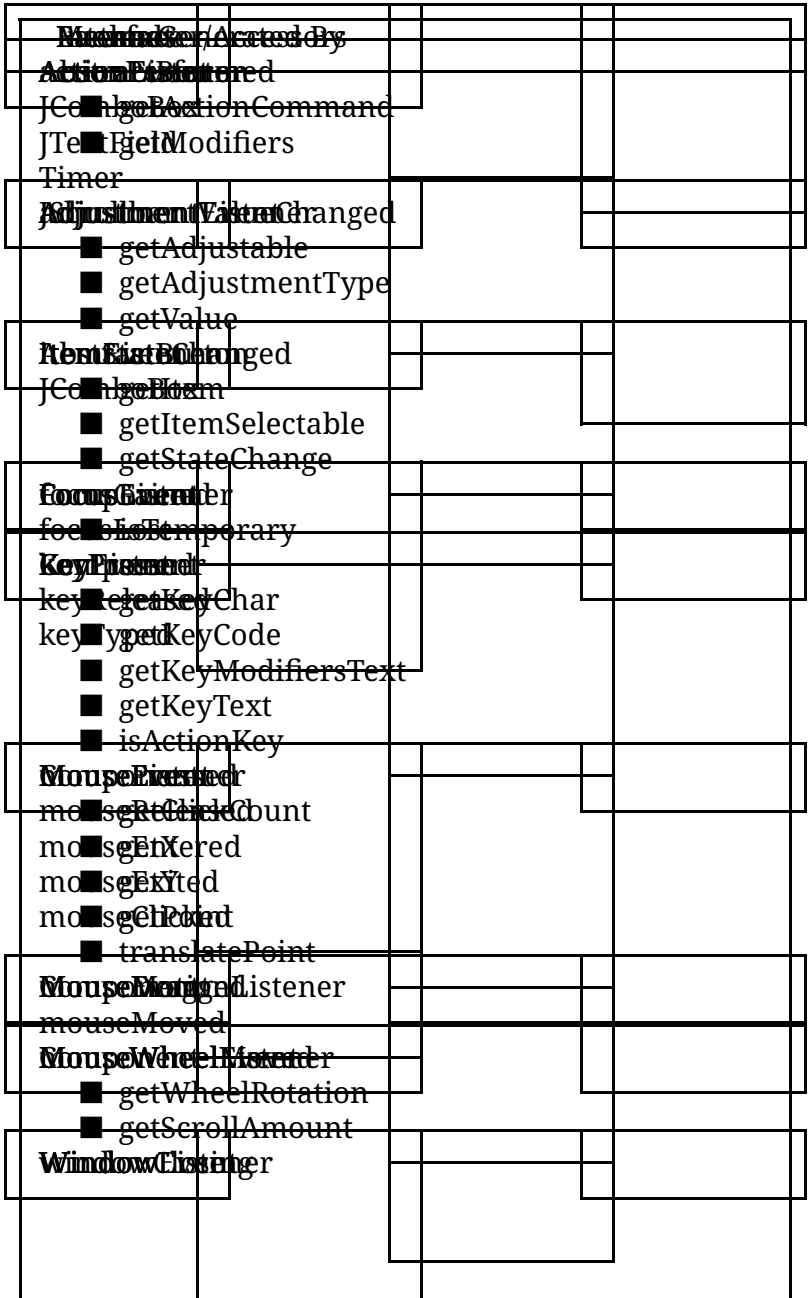
- ActionEvent (for a button click, a menu selection, selecting a list item, or Enter typed in a text field)
- AdjustmentEvent (the user adjusted a scrollbar)
- ItemEvent (the user made a selection from a set of checkbox or list items)

Five low-level event classes are commonly used:

- KeyEvent (a key was pressed or released)
- MouseEvent (the mouse button was pressed, released, moved, or dragged)

- MouseWheelEvent (the mouse wheel was rotated)
- FocusEvent (a component got focus or lost focus)
- WindowEvent (the window state changed)

Table 10.4 shows the most important AWT listener interfaces, events, and event sources.



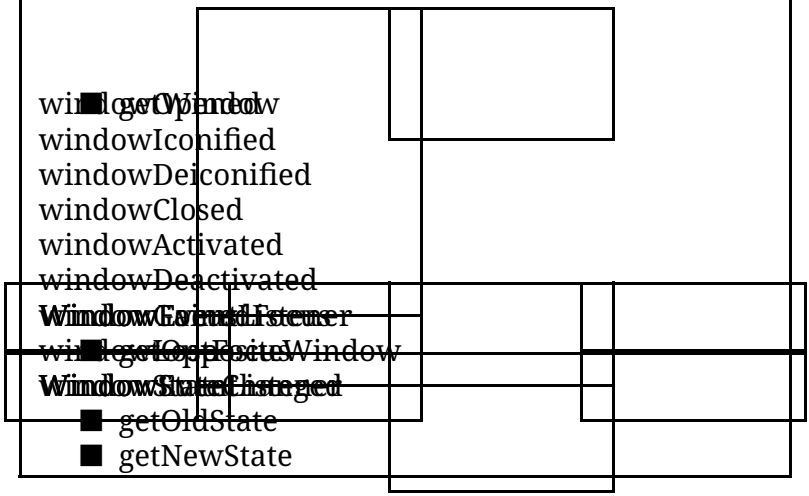


Table 10.4: Event Handling Summary

10.5. The Preferences API

I end this chapter with a discussion of the java.util.prefs API. In a desktop program, you will often want to store user preferences, such as the last file that the user worked on, the last window location, and so on.

As you have seen in Chapter 9 of Volume I, the Properties class makes it simple to load and save configuration information of a program. However, using property files has these disadvantages:

- Some operating systems have no concept of a home directory, making it difficult to find a uniform location for configuration files.
- There is no standard convention for naming configuration files, increasing the likelihood of name clashes as users install multiple Java applications.

Some operating systems have a central repository for configuration information. The best-known example is the registry in Microsoft Windows. The Preferences class provides such a central repository in a platform-independent manner. In Windows, the Preferences class uses the registry for storage; on Linux, the information is stored in the local file system instead. Of course, the repository implementation is transparent to the

programmer using the Preferences class.

The Preferences repository has a tree structure, with node path names such as /com/mycompany/myapp. As with package names, name clashes are avoided as long as programmers start the paths with reversed domain names. In fact, the designers of the API suggest that the configuration node paths match the package names in your program.

Each node in the repository has a separate table of key/value pairs that you can use to store numbers, strings, or byte arrays. No provision is made for storing serializable objects. The API designers felt that the serialization format is too fragile for long-term storage. Of course, if you disagree, you can save serialized objects in byte arrays.

For additional flexibility, there are multiple parallel trees. Each program user has one tree; an additional tree, called the system tree, is available for settings that are common to all users. The Preferences class uses the operating system's notion of the "current user" for accessing the appropriate user tree.

To access a node in the tree, start with the user or system root:

```
Preferences root = Preferences.userRoot();
```

or

```
Preferences root = Preferences.systemRoot();
```

Then access the node. You can simply provide a node path name:

```
Preferences node = root.node("/com/mycompany/myapp");
```

A convenient shortcut gets a node whose path name equals the package name of a class. Simply take an object of that class and call

```
Preferences node = Preferences.userNodeForPackage(obj.getClass());
```

or

```
Preferences node = Preferences.systemNodeForPackage(obj.getClass
```

Typically, obj will be the this reference.

Once you have a node, you can access the key/value table with methods

```
String get(String key, String def)
int getInt(String key, int def)
long getLong(String key, long def)
float getFloat(String key, float def)
double getDouble(String key, double def)
boolean getBoolean(String key, boolean def)
byte[] getByteArray(String key, byte[] def)
```

Note that you must specify a default value when reading the information, in case the repository data is not available. Defaults are required for several reasons. The data might be missing because the user never specified a preference. Certain resource-constrained platforms might not have a repository, and mobile devices might be temporarily disconnected from the repository.

Conversely, you can write data to the repository with put methods such as

```
put(String key, String value)
putInt(String key, int value)
```

and so on.

You can enumerate all keys stored in a node with the method

```
String[] keys()
```

There is currently no way to find out the type of the value of a particular key.

> Node names and keys are limited to 80 characters, and string values to 8,192 characters.

Central repositories such as the Windows registry traditionally suffer from two problems:

- They turn into a “dumping ground” filled with obsolete information.
- Configuration data gets entangled into the repository, making it difficult to move preferences to a new platform.

The Preferences class has a solution for the second problem. You can export the preferences of a subtree (or, less commonly, a single node) by calling the methods

```
void exportSubtree(OutputStream out)
void exportNode(OutputStream out)
```

The data are saved in XML format. You can import them into another repository by calling

```
void importPreferences(InputStream in)
```

Here is a sample file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preference
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="com">
      <map/>
      <node name="horstmann">
        <map/>
        <node name="corejava">
          <map>
        <entry key="height" value="200.0"/>
        <entry key="left" value="1027.0"/>
        <entry key="filename" value="/home/cay/books/cj11/code/v1cl
        <entry key="top" value="380.0"/>
        <entry key="width" value="300.0"/>
          </map>
        </node>
      </node>
```

```
      </node>
    </root>
  </preferences>
```

If your program uses preferences, you should give your users the opportunity of exporting and importing them, so they can easily migrate their settings from one computer to another. The program in Listing 10.8 demonstrates this technique. The program simply saves the window location and the last loaded filename. Try resizing the window, then export your preferences, move the window, exit, and restart the application. The window will be just like you left it when you exited. Import your preferences, and the window reverts to its prior location.

Listing 10.8 **v2ch10/preferences/ImageViewer.java**

```
package v2ch10.preferences;

import module java.base;
import module java.desktop;
import module java.prefs;

/**
 * A program to test preference settings. The program remembers the frame po
 * last selected file.
 */
class ImageViewer {
   void main() {
      EventQueue.invokeLater(() -> {
         var frame = new ImageViewerFrame();
         frame.setTitle("ImageViewer");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * An image viewer that restores position, size, and image from user preferenc
 * the preferences upon exit.
```

```
 */
class ImageViewerFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;
   private String image;

   public ImageViewerFrame() {
      Preferences root = Preferences.userRoot();
      Preferences node = root.node("/com/horstmann/corejava/ImageViewer");
      // get position, size, title from properties
      int left = node.getInt("left", 0);
      int top = node.getInt("top", 0);
      int width = node.getInt("width", DEFAULT_WIDTH);
      int height = node.getInt("height", DEFAULT_HEIGHT);
      setBounds(left, top, width, height);
      image = node.get("image", null);
      var label = new JLabel();
      if (image != null) label.setIcon(new ImageIcon(image));

      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent event) {
            node.putInt("left", getX());
            node.putInt("top", getY());
            node.putInt("width", getWidth());
            node.putInt("height", getHeight());
            if (image != null) node.put("image", image);
         }
      });

      // use a label to display the images
      add(label);

      // set up the file chooser
      var chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      // set up the menu bar
      var menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      var menu = new JMenu("File");
      menuBar.add(menu);

      var openItem = new JMenuItem("Open");
```

```
      menu.add(openItem);
      openItem.addActionListener(_ -> {
         // show file chooser dialog
         int result = chooser.showOpenDialog(null);

         // if file selected, set it as icon of the label
         if (result == JFileChooser.APPROVE_OPTION) {
            image = chooser.getSelectedFile().getPath();
            label.setIcon(new ImageIcon(image));
         }
      });

      var exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(_ -> System.exit(0));
   }
}
```

java.util.prefs.Preferences 1.4

- Preferences userRoot()
  returns the root preferences node of the user of the calling program.
- Preferences systemRoot()
  returns the systemwide root preferences node.
- Preferences node(String path)
  returns a node that can be reached from the current node by the given path. If path is absolute (that is, starts with a /), then the node is located starting from the root of the tree containing this preference node. If there isn't a node with the given path, it is created.
- Preferences userNodeForPackage(Class cl)
- Preferences systemNodeForPackage(Class cl)
  return a node in the current user's tree or the system tree whose absolute node path corresponds to the package name of the class cl.
- String[] keys()
  returns all keys belonging to this node.
- String get(String key, String def)
- int getInt(String key, int def)
- long getLong(String key, long def)
- float getFloat(String key, float def)

- double getDouble(String key, double def)
- boolean getBoolean(String key, boolean def)
- byte[] getByteArray(String key, byte[] def)
  return the value associated with the given key—or the supplied default value if no value is associated with the key, the associated value is not of the correct type, or the preferences store is unavailable.
- void put(String key, String value)
- void putInt(String key, int value)
- void putLong(String key, long value)
- void putFloat(String key, float value)
- void putDouble(String key, double value)
- void putBoolean(String key, boolean value)
- void putByteArray(String key, byte[] value)
  store a key/value pair with this node.
- void exportSubtree(OutputStream out)
  writes the preferences of this node and its children to the specified stream.
- void exportNode(OutputStream out)
  writes the preferences of this node (but not its children) to the specified stream.
- void importPreferences(InputStream in)
  imports the preferences contained in the specified stream.

This concludes our introduction into graphical user interface programming. The next chapter shows you how to work with the most common Swing components.

Chapter 11 ■ User Interface Components with Swing

The previous chapter was written primarily to show you how to use the event model in Java. In the process, you took the first steps toward learning how to build a graphical user interface. This chapter shows you the most important tools you'll need to build more full-featured GUIs.

We start out with a tour of the architectural underpinnings of Swing. Knowing what goes on "under the hood" is important in understanding how to use some of the more advanced components effectively. I then show you the most common user interface components in Swing, such as text fields, radio buttons, and menus. Next, you will learn how to use layout managers to arrange these components. Finally, you'll see how to implement dialog boxes in Swing.

This chapter covers the basic Swing components such as text components, buttons, and sliders. These are the essential user interface components that you will need most frequently. We will cover advanced Swing components in Chapter 12.

11.1. Swing and the Model-View-Controller Design Pattern

Let's step back for a minute and think about the pieces that make up a user interface component such as a button, a checkbox, a text field, or a sophisticated tree control. Every component has three characteristics:

- Its content, such as the state of a button (pushed in or not), or the text in a text field
- Its visual appearance (color, size, and so on)
- Its behavior (reaction to events)

Even a seemingly simple component such as a button exhibits some moderately complex interaction among these characteristics. Obviously, the visual appearance of a button depends on the look-and-feel. A Metal button looks different from a Windows button or a Motif button. In addition, the appearance depends on the button state; when a button is pushed in, it needs to be redrawn to look different. The state depends on the events that the button receives. When the user depresses the mouse inside the button, the button is pushed in.

Of course, when you use a button in your programs, you simply consider it as a button; you don't think too much about the inner workings and characteristics. That, after all, is the job of the programmer who implemented the button. However, programmers who implement buttons and all other user interface components are motivated to think a little harder about them, so that they work well no matter what look-and-feel is in effect.

To do this, the Swing designers turned to a well-known design pattern: the model-view-controller (MVC) pattern. This design pattern tells us to provide three separate objects:

- The model, which stores the content
- The view, which displays the content
- The controller, which handles user input

The pattern specifies precisely how these three objects interact. The model stores the content and has no user interface. For a button, the content is pretty trivial—just a small set of flags that tells whether the button is currently pushed in or out, whether it is active or inactive, and so on. For a text field, the content is a bit more interesting. It is a string object that holds the current text. This is not the same as the view of the content—if the content is larger than the text field, the user sees only a portion of the text displayed (see Figure 11.1).

.

Figure 11.1: Model and view of a text field

The model must implement methods to change the content and to discover what the content is. For example, a text model has methods to add or remove characters in the current text and to return the current text as a string. Again, keep in mind that the model is completely nonvisual. It is the job of a view to draw the data stored in the model.

> The term "model" is perhaps unfortunate because we often think of a model as a representation of an abstract concept. Car and airplane designers build models to simulate real cars and planes. But that analogy really leads you astray when thinking about the model-view-controller pattern. In this design pattern, the model stores the complete content, and the view gives a (complete or incomplete) visual representation of the content. A better analogy might be the model who poses for an artist. It is up to the artist to look at the model and create a view. Depending on the artist, that view might be a formal portrait, an impressionist painting, or a cubist drawing with strangely contorted limbs.

One of the advantages of the model-view-controller pattern is that a model can have multiple views, each showing a different part or aspect of the full content. For example, an HTML editor

can offer two simultaneous views of the same content: a WYSIWYG view and a “raw tag” view (see Figure 11.2). When the model is updated through the controller of one of the views, it tells both attached views about the change. When the views are notified, they refresh themselves automatically. Of course, for a simple user interface component such as a button, you won’t have multiple views of the same model.

.

Figure 11.2: Two separate views of the same model

The controller handles the user-input events, such as mouse clicks and keystrokes. It then decides whether to translate these events into changes in the model or the view. For example, if the user presses a character key in a text box, the controller calls the “insert character” command of the model. The model then tells the view to update itself. The view never knows why the text changed. But if the user presses a cursor key, the controller may tell the view to scroll. Scrolling the view has no effect on the underlying text, so the model never knows that this event happened.

Figure 11.3 shows the interactions among model, view, and controller objects.

.

Figure 11.3: Interactions among model, view, and controller objects

For most Swing components, the model class implements an interface whose name ends in Model; in this case, the interface is called ButtonModel. Classes implementing that interface can define the state of the various kinds of buttons. Actually, buttons aren’t all that complicated, and the Swing library contains a single class, called DefaultButtonModel, that implements this interface.

You can get a sense of the sort of data maintained by a button model by looking at the properties of the ButtonModel interface —see Table 11.1.

Property Name	Value
actionCommand	The action command string associated with this button
mnemonic	The keyboard mnemonic for this button
armed	true if the button was pressed and the mouse is still over the button
enabled	true if the button is selectable
pressed	true if the button was pressed but the mouse button hasn't yet been released
rollover	true if the mouse is over the button
selected	true if the button has been toggled on (used for checkboxes and radio buttons)

Table 11.1: Properties of the ButtonModel Interface

Each JButton object stores a button model object which you can retrieve.

```
var button = new JButton("Blue");
ButtonModel model = button.getModel();
```

In practice, you won't care—the minutiae of the button state are only of interest to the view that draws it. All the important information—such as whether a button is enabled—is available from the JButton class. (Of course, the JButton then asks its model to retrieve that information.)

Have another look at the ButtonModel interface to see what isn't there. The model does not store the button label or icon. There is no way to find out what's on the face of a button just by looking at its model. (Actually, as you will see in Section 11.4.2, this purity of design is the source of some grief for the programmer.)

It is also worth noting that the same model (namely, DefaultButtonModel) is used for push buttons, radio buttons, checkboxes, and even menu items. Of course, each of these

button types has different views and controllers. When using the Metal look-and-feel, the JButton uses a class called BasicButtonUI for the view and a class called ButtonUIListener as controller. In general, each Swing component has an associated view object that ends in UI. But not all Swing components have dedicated controller objects.

So, having read this short introduction to what is going on under the hood in a JButton, you may be wondering: Just what is a JButton really? It is simply a wrapper class inheriting from JComponent that holds the DefaultButtonModel object, some view data (such as the button label and icons), and a BasicButtonUI object that is responsible for the button view and installs a BasicButtonListener with button-specific controller functionality.

11.2. Introduction to Layout Management

Before we go on to discussing individual Swing components, such as text fields and radio buttons, let's briefly cover how to arrange these components inside a frame.

Of course, Java development environments have drag-and-drop GUI builders. Nevertheless, it is important to know exactly what goes on “under the hood” because even the best of these tools will usually require hand-tweaking.

11.2.1. Layout Managers

Let’s start by reviewing the program from Chapter 10 that used buttons to change the background color of a frame.

The buttons are contained in a JPanel object and are managed by the flow layout manager, the default layout manager for a panel. Figure 11.4 shows what happens when you add more buttons to the panel. As you can see, a new row is started when there is no more room.

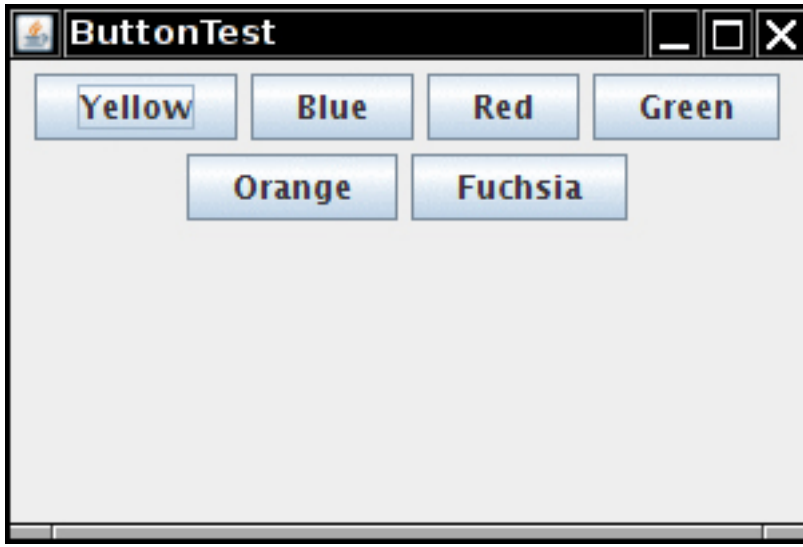


Figure 11.4: A panel with six buttons managed by a flow layout

Moreover, the buttons stay centered in the panel, even when the user resizes the frame (see Figure 11.5).

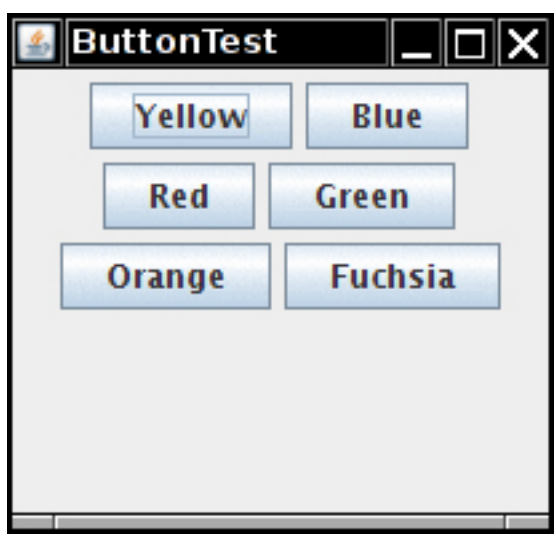


Figure 11.5: Changing the panel size rearranges the buttons automatically.

In general, components are placed inside containers, and a layout manager determines the positions and sizes of components in a container.

Buttons, text fields, and other user interface elements extend the class Component. Components can be placed inside containers, such as panels. Containers can themselves be put inside other containers, so the class Container extends Component. Figure 11.6 shows the inheritance hierarchy for Component.

.

Figure 11.6: Inheritance hierarchy for the Component class

Unfortunately, the inheritance hierarchy is somewhat unclean in two respects. First, top-level windows, such as JFrame, are subclasses of Container and hence Component, but they cannot be placed inside other containers. Moreover, JComponent is a subclass of Container, not Component. Therefore one can add other components into a JButton. (However, those components would not be displayed.)

Each container has a default layout manager, but you can always set your own. For example, the statement

```
panel.setLayout(new GridLayout(4, 4));
```

uses the GridLayout class to lay out the components in four rows and four columns. When you add components to the container, the add method of the container passes the component and any placement directions to the layout manager.

java.awt.Container 1.0
■ void setLayout(LayoutManager m)

sets the layout manager for this container.

- Component add(Component c)
- Component add(Component c, Object constraints)1.1
  add a component to this container and return the component reference.

java.awt.FlowLayout1.0

- FlowLayout()
- FlowLayout(int align)
- FlowLayout(int align, int hgap, int vgap)
  construct a new FlowLayout. The align parameter is one of LEFT, CENTER, or RIGHT.

11.2.2. Border Layout

The border layout manager is the default layout manager of the content pane of every JFrame. Unlike the flow layout manager, which completely controls the position of each component, the border layout manager lets you choose where you want to place each component. You can choose to place the component in the center, north, south, east, or west of the content pane (see Figure 11.7).

.

Figure 11.7: Border layout

For example:

```
frame.add(component, BorderLayout.SOUTH);
```

The edge components are laid out first, and the remaining available space is occupied by the center. When the container is resized, the dimensions of the edge components are unchanged, but the center component changes its size. Add components by specifying a constant CENTER, NORTH, SOUTH,

EAST, or WEST of the BorderLayout class. Not all of the positions need to be occupied. If you don’t supply any value, CENTER is assumed.

The BorderLayout constants are defined as strings. For example, BorderLayout.SOUTH is defined as the string "South". This is safer than using strings. If you accidentally misspell a string, for example, frame.add(component, "south"), the compiler won’t catch that error.

Unlike the flow layout, the border layout grows all components to fill the available space. (The flow layout leaves each component at its preferred size.) This is a problem when you add a button:

```
frame.add(yellowButton, BorderLayout.SOUTH); // don't
```

Figure 11.8 shows what happens when you use the preceding code fragment. The button has grown to fill the entire southern region of the frame. And, if you were to add another button to the southern region, it would just displace the first button.



Figure 11.8: A single button managed by a border layout

To solve this problem, use additional panels. For example, look at Figure 11.9. The three buttons at the bottom of the screen are all contained in a panel. The panel is put into the southern region of the content pane.

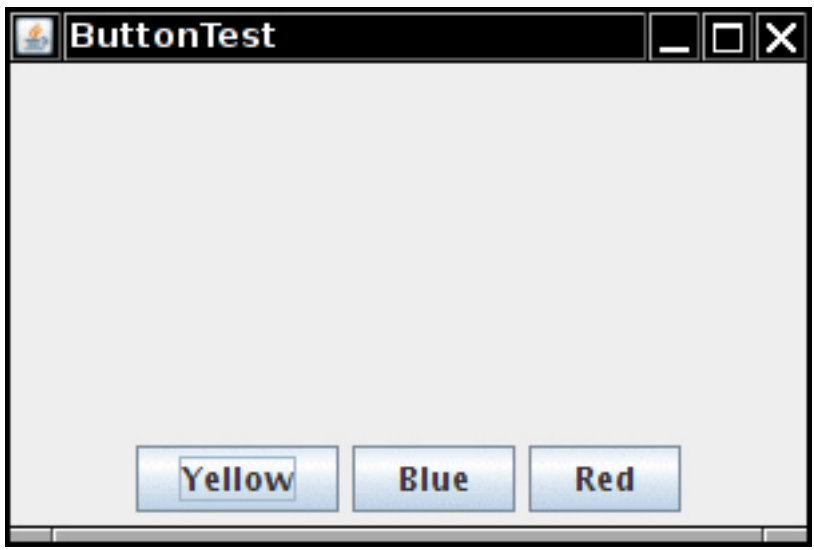


Figure 11.9: Panel placed in the southern region of the frame

To achieve this configuration, first create a new JPanel object, then add the individual buttons to the panel. The default layout manager for a panel is a FlowLayout, which is a good choice for this situation. Add the individual buttons to the panel, using the add method you have seen before. The position and size of the buttons is under the control of the FlowLayout manager. This means the buttons stay centered within the panel and do not expand to fill the entire panel area. Finally, add the panel to the content pane of the frame.

```
var panel = new JPanel();
panel.add(yellowButton);
```

```
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);
```

The border layout expands the size of the panel to fill the entire southern region.

java.awt.BorderLayout1.0
■ BorderLayout() ■ BorderLayout(int hgap, int vgap) construct a new BorderLayout.

11.2.3. Grid Layout

The grid layout arranges all components in rows and columns like a spreadsheet. All components are given the same size. The calculator program in Figure 11.10 uses a grid layout to arrange the calculator buttons. When you resize the window, the buttons grow and shrink, but all buttons have identical sizes.

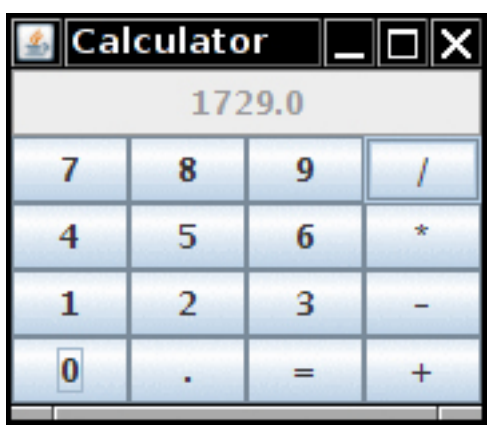


Figure 11.10: A calculator

In the constructor of the grid layout object, you specify how

many rows and columns you need.

```
panel.setLayout(new GridLayout(4, 4));
```

Add the components, starting with the first entry in the first row, then the second entry in the first row, and so on.

```
panel.add(new JButton("1"));
panel.add(new JButton("2"));
```

Of course, few applications have as rigid a layout as the face of a calculator. In practice, small grids (usually with just one row or one column) can be useful to organize partial areas of a window. For example, if you want to have a row of buttons of identical sizes, you can put the buttons inside a panel that is governed by a grid layout with a single row.

java.awt.GridLayout1.0

- GridLayout(int rows, int columns)
- GridLayout(int rows, int columns, int hgap, int vgap)
  construct a new GridLayout. One of rows and columns (but not both) may be zero, denoting an arbitrary number of components per row or column.

11.3. Text Input

We are finally ready to start introducing the Swing user interface components. Let's begin with the components that let a user input and edit text. You can use the JTextField and JTextArea components for text input. A text field can accept only one line of text; a text area can accept multiple lines of text. A JPasswordField accepts one line of text without showing the contents.

All three of these classes inherit from a class called JTextComponent. You will not be able to construct a JTextComponent yourself because it is an abstract class. On the

other hand, as is so often the case in Java, when you go searching through the API documentation, you may find that the methods you are looking for are actually in the parent class JTextComponent rather than the derived class. For example, the methods that get or set the text in a text field or text area are actually in JTextComponent.

javax.swing.text.JTextComponent1.2

- String getText()
- void setText(String text)
  get or set the text of this text component.
- boolean isEditable()
- void setEditable(boolean b)
  get or set the editable property that determines whether the user can edit the content of this text component.

11.3.1. Text Fields

The usual way to add a text field to a window is to add it to a panel or other container—just as you would add a button:

```
var panel = new JPanel();
var textField = new JTextField("Default input", 20);
panel.add(textField);
```

This code adds a text field and initializes it by placing the string "Default input" inside it. The second parameter of this constructor sets the width. In this case, the width is 20 “columns.” Unfortunately, a column is a rather imprecise measurement. One column is the expected width of one character in the font you are using for the text. The idea is that if you expect the inputs to be n characters or less, you are supposed to specify n as the column width. In practice, this measurement doesn’t work out too well, and you should add 1 or 2 to the maximum input length to be on the safe side. Also, keep in mind that the number of columns is only a hint to the AWT that gives the preferred size. If the layout manager needs

to grow or shrink the text field, it can adjust its size. The column width that you set in the JTextField constructor is not an upper limit on the number of characters the user can enter. The user can still type in longer strings, but the input scrolls when the text exceeds the length of the field. Users tend to find scrolling text fields irritating, so you should size the fields generously. If you need to reset the number of columns at runtime, you can do that with the setColumns method.

After changing the size of a text box with the setColumns method, call the revalidate method of the surrounding container.

```
textField.setColumns(10);
panel.revalidate();
```

The revalidate method recomputes the size and layout of all components in a container. After you use the revalidate method, the layout manager resizes the container, and the changed size of the text field will be visible.

The revalidate method belongs to the JComponent class. It doesn't immediately resize the component but merely marks it for resizing. This approach avoids repetitive calculations if multiple components request to be resized. However, if you want to recompute all components inside a JFrame, you have to call the validate method—JFrame doesn't extend JComponent.

In general, users add text (or edit an existing text) in a text field. Quite often these text fields start out blank. To make a blank text field, just don't provide a string argument for the JTextField constructor:

```
var textField = new JTextField(20);
```

You can change the content of the text field at any time by using the setText method from the JTextComponent parent class mentioned in the previous section. For example:

```
textField.setText("Hello!");
```

And, as was mentioned in the previous section, you can find out what the user typed by calling the getText method. This method returns the exact text that the user has typed. To strip any extraneous leading and trailing spaces from the data in a text field, apply the strip method to the return value of getText:

```
String text = textField.getText().strip();
```

To change the font in which the user text appears, use the setFont method.

javax.swing.JTextField1.2

- JTextField(int columns)
  constructs an empty JTextField with the specified number of columns.
- JTextField(String text, int columns)
  constructs a new JTextField with an initial string and the specified number of columns.
- int getColumns()
- void setColumns(int columns)
  get or set the number of columns that this text field should use.

javax.swing.JComponent1.2

- void revalidate()
  causes the position and size of a component to be recomputed.
- void setFont(Font f)
  sets the font of this component.

java.awt.Component1.0

- void validate()

recomputes the position and size of a component. If the component is a container, the positions and sizes of its components are recomputed.

- Font getFont()
  gets the font of this component.

11.3.2. Labels and Labeling Components

Labels are components that hold text. They have no decorations (for example, no boundaries). They also do not react to user input. You can use a label to identify components. For example, unlike buttons, text fields have no label to identify them. To label a component that does not itself come with an identifier:

1. Construct a JLabel component with the correct text.
2. Place it close enough to the component you want to identify so that the user can see that the label identifies the correct component.

The constructor for a JLabel lets you specify the initial text or icon and, optionally, the alignment of the content. Use constants from the SwingConstants interface to specify alignment. That interface defines a number of useful constants such as LEFT, RIGHT, CENTER, NORTH, EAST, and so on. The JLabel class is one of several Swing classes that implement this interface. Therefore, you can specify a right-aligned label either as

```
var label = new JLabel("User name: ", SwingConstants.RIGHT);
```

or

```
var label = new JLabel("User name: ", JLabel.RIGHT);
```

The setText and setIcon methods let you set the text and icon of the label at runtime.

You can use both plain and HTML text in buttons, labels,

and menu items. I don't recommend HTML in buttons—it interferes with the look-and-feel. But HTML in labels can be very effective. Simply surround the label string with <html>. . .</html>, like this:

```
label = new JLabel("<html><b>Required</b> entry:</html>");
```

Note that the first component with an HTML label may take some time to be displayed because the rather complex HTML rendering code must be loaded.

Labels can be positioned inside a container like any other component. This means you can use the techniques you have seen before to place your labels where you need them.

javax.swing.JLabel 1.2

- JLabel(String text)
- JLabel(Icon icon)
- JLabel(String text, int align)
- JLabel(String text, Icon icon, int align)
  construct a label. The align parameter is one of the SwingConstants constants LEFT (default), CENTER, or RIGHT.
- String getText()
- void setText(String text)
  get or set the text of this label.
- Icon getIcon()
- void setIcon(Icon icon)
  get or set the icon of this label.

11.3.3. Password Fields

Password fields are a special kind of text fields. To prevent nosy bystanders from seeing your password, the characters that the user enters are not actually displayed. Instead, each typed character is represented by an echo character, such as a bullet (•). Swing supplies a JPasswordField class that implements such

a text field.

The password field is another example of the power of the model-view-controller architecture pattern. The password field uses the same model to store the data as a regular text field, but its view has been changed to display all characters as echo characters.

javax.swing.JPasswordField1.2

- JPasswordField(String text, int columns)
  constructs a new password field.
- void setEchoChar(char echo)
  sets the echo character for this password field. This is advisory; a particular look-and-feel may insist on its own choice of echo character. A value of 0 resets the echo character to the default.
- char[] getPassword()
  returns the text contained in this password field. For stronger security, you should overwrite the content of the returned array after use. (The password is not returned as a String because a string would stay in the virtual machine until it is garbage-collected.)

11.3.4. Text Areas

Sometimes, you need to collect user input that is more than one line long. As mentioned earlier, you can use the JTextArea component for this. When you place a text area component in your program, a user can enter any number of lines of text, using the Enter key to separate them. Each line ends with a '\n' (even on Windows). Figure 11.11 shows a text area at work.

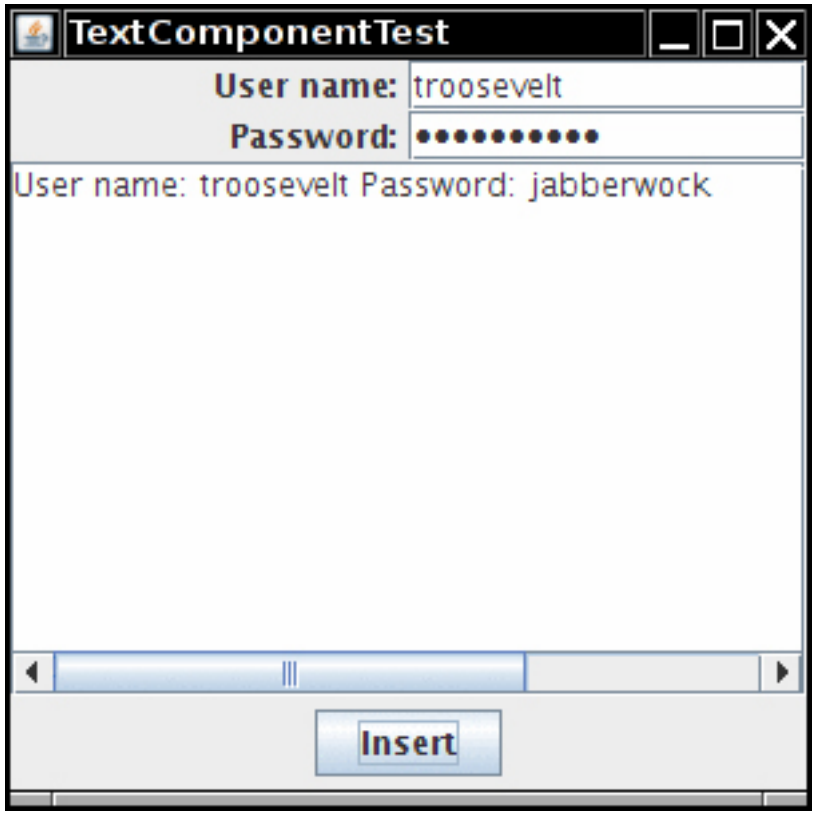


Figure 11.11: Text components

In the constructor for the JTextArea component, specify the number of rows and columns for the text area. For example,

```
textArea = new JTextArea(8, 40); // 8 lines of 40 columns each
```

where the columns parameter works as before—and you still need to add a few more columns for safety's sake. Also, as before, the user is not restricted to the number of rows and columns; the text simply scrolls when the user inputs too much. You can also use the setColumns method to change the number of columns and the setRows method to change the number of rows. These numbers only indicate the preferred size—the layout manager can still grow or shrink the text area.

If there is more text than the text area can display, the

remaining text is simply clipped. You can avoid clipping long lines by turning on line wrapping:

```
textArea.setLineWrap(true); // long lines are wrapped
```

This wrapping is a visual effect only; the text in the document is not changed—no automatic '\n' characters are inserted into the text.

11.3.5. Scroll Panes

In Swing, a text area does not have scrollbars. If you want scrollbars, you have to place the text area inside a scroll pane.

```
textArea = new JTextArea(8, 40);
var scrollPane = new JScrollPane(textArea);
```

The scroll pane now manages the view of the text area. Scrollbars automatically appear if there is more text than the text area can display, and they vanish again if text is deleted and the remaining text fits inside the area. The scrolling is handled internally by the scroll pane—your program does not need to process scroll events.

This is a general mechanism that works for any component, not just text areas. To add scrollbars to a component, put them inside a scroll pane.

Listing 11.1 demonstrates the various text components. This program shows a text field, a password field, and a text area with scrollbars. The text field and password field are labeled. Click on “Insert” to insert the field contents into the text area.

The JTextArea component displays plain text only, without special fonts or formatting. To display formatted text (such as HTML), you can use the JEditorPane class that is discussed in Chapter 12.

Listing 11.1 **v2ch11/text/TextComponentFrame.java**

```
package v2ch11.text;

import module java.desktop;

/**
 * A frame with sample text components.
 */
public class TextComponentFrame extends JFrame {
   private static final int TEXTAREA_ROWS = 8;
   private static final int TEXTAREA_COLUMNS = 20;

   public TextComponentFrame() {
      var textField = new JTextField();
      var passwordField = new JPasswordField();

      var northPanel = new JPanel();
      northPanel.setLayout(new GridLayout(2, 2));
      northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
      northPanel.add(textField);
      northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
      northPanel.add(passwordField);

      add(northPanel, BorderLayout.NORTH);

      var textArea = new JTextArea(TEXTAREA_ROWS, TEXTAREA_COLUMNS);
      var scrollPane = new JScrollPane(textArea);

      add(scrollPane, BorderLayout.CENTER);

      // add button to append text into the text area

      var southPanel = new JPanel();

      var insertButton = new JButton("Insert");
      southPanel.add(insertButton);
      insertButton
            .addActionListener(_ -> textArea.append("User name: " + textField.get
                  + " Password: " + new String(passwordField.getPassword()) + "\n"

      add(southPanel, BorderLayout.SOUTH);
      pack();
   }
```

}

javax.swing.JTextArea1.2

- JTextArea()
- JTextArea(int rows, int columns)
- JTextArea(String text, int rows, int columns)
  construct a new text area.
- void setColumns(int columns)
  tells the text area the preferred number of columns it should use.
- void setRows(int rows)
  tells the text area the preferred number of rows it should use.
- void append(String newText)
  appends the given text to the end of the text already in the text area.
- void setLineWrap(boolean wrap)
  turns line wrapping on or off.
- void setWrapStyleWord(boolean word)
  If word is true, long lines are wrapped at word boundaries. If it is false, long lines are broken without taking word boundaries into account.
- void setTabSize(int c)
  sets tab stops every c columns. Note that the tabs aren't converted to spaces but cause alignment with the next tab stop.

javax.swing.JScrollPane1.2

- JScrollPane(Component c)
  creates a scroll pane that displays the content of the specified component. Scrollbars are supplied when the component is larger than the view.

11.4. Choice Components

You now know how to collect text input from users, but there are many occasions where you would rather give users a finite set of choices than have them enter the data in a text component. Using a set of buttons or a list of items tells your users what choices they have. (It also saves you the trouble of error checking.) In this section, you will learn how to program checkboxes, radio buttons, lists of choices, and sliders.

11.4.1. Checkboxes

If you want to collect just a “yes” or “no” input, use a checkbox component. Checkboxes automatically come with labels that identify them. The user can check the box by clicking inside it and turn off the checkmark by clicking inside the box again. Pressing the space bar when the focus is in the checkbox also toggles the checkmark.

Figure 11.12 shows a simple program with two checkboxes, one for turning the italic attribute of a font on or off, and the other for boldface. Note that the second checkbox has focus, as indicated by the rectangle around the label. Each time the user clicks one of the checkboxes, the screen is refreshed, using the new font attributes.

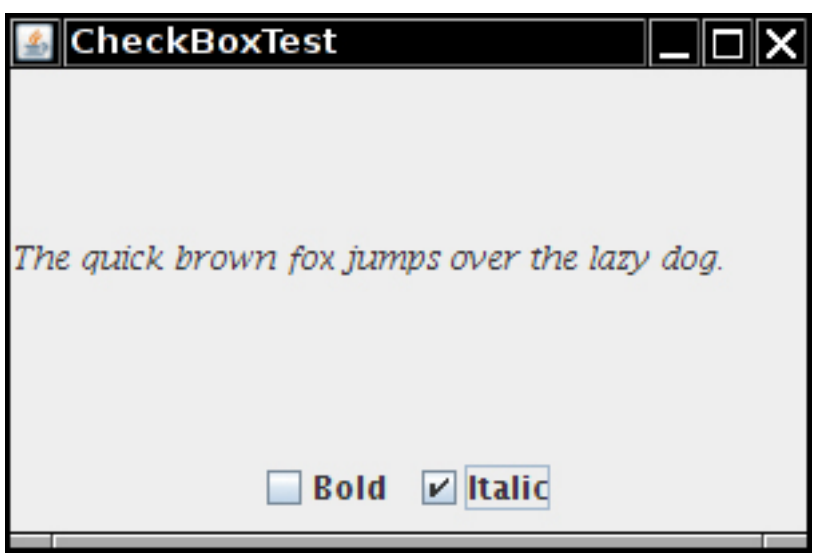


Figure 11.12: Checkboxes

Checkboxes need a label next to them to identify their purpose. Give the label text in the constructor:

```
bold = new JCheckBox("Bold");
```

Use the setSelected method to turn a checkbox on or off. For example:

```
bold.setSelected(true);
```

The isSelected method then retrieves the current state of each checkbox. It is false if unchecked, true if checked.

When the user clicks on a checkbox, this triggers an action event. As always, you attach an action listener to the checkbox. In our program, the two checkboxes share the same action listener.

```
ActionListener listener = . . .;
bold.addActionListener(listener);
italic.addActionListener(listener);
```

The listener queries the state of the bold and italic checkboxes and sets the font of the panel to plain, bold, italic, or both bold and italic.

```
ActionListener listener = _ -> {
   int mode = 0;
   if (bold.isSelected()) mode += Font.BOLD;
   if (italic.isSelected()) mode += Font.ITALIC;
   label.setFont(new Font(Font.SERIF, mode, FONTSIZE));
};
```

Listing 11.2 is the program listing for the checkbox example.

Listing 11.2 **v2ch11/checkBox/CheckBoxFrame.java**

```
package v2ch11.checkBox;

import module java.desktop;

/**
 * A frame with a sample text label and check boxes for selecting font attribute
 */
public class CheckBoxFrame extends JFrame {
   private JLabel label;
   private JCheckBox bold;
   private JCheckBox italic;
   private static final int FONTSIZE = 24;

   public CheckBoxFrame() {
      // add the sample text label

      label = new JLabel("The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.BOLD, FONTSIZE));
      add(label, BorderLayout.CENTER);

      // this listener sets the font attribute of
      // the label to the check box state

      ActionListener listener = _ -> {
         int mode = 0;
         if (bold.isSelected()) mode += Font.BOLD;
         if (italic.isSelected()) mode += Font.ITALIC;
         label.setFont(new Font("Serif", mode, FONTSIZE));
      };

      // add the check boxes
      var buttonPanel = new JPanel();

      bold = new JCheckBox("Bold");
      bold.addActionListener(listener);
      bold.setSelected(true);
      buttonPanel.add(bold);

      italic = new JCheckBox("Italic");
      italic.addActionListener(listener);
      buttonPanel.add(italic);

      add(buttonPanel, BorderLayout.SOUTH);
      pack();
```

```
   }
}
```

javax.swing.JCheckBox1.2

- JCheckBox(String label)
- JCheckBox(String label, Icon icon)
  construct a checkbox that is initially unselected.
- JCheckBox(String label, boolean state)
  constructs a checkbox with the given label and initial state.
- boolean isSelected()
- void setSelected(boolean state)
  get or set the selection state of the checkbox.

11.4.2. Radio Buttons

In the previous example, the user could check either, both, or neither of the two checkboxes. In many cases, we want the user to check only one of several boxes. When another box is checked, the previous box is automatically unchecked. Such a group of boxes is often called a radio button group because the buttons work like the station selector buttons on a radio. When you push in one button, the previously depressed button pops out. Figure 11.13 shows a typical example. We allow the user to select a font size from among the choices—Small, Medium, Large, or Extra large—but, of course, we will allow selecting only one size at a time.



Figure 11.13: A radio button group

Implementing radio button groups is easy in Swing. You construct one object of type ButtonGroup for every group of buttons. Then, you add objects of type JRadioButton to the button group. The button group object is responsible for turning off the previously set button when a new button is clicked.

```
var group = new ButtonGroup();

var smallButton = new JRadioButton("Small", false);
group.add(smallButton);

var mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
. . .
```

The second argument of the constructor is true for the button that should be checked initially and false for all others. Note that the button group controls only the behavior of the buttons; if you want to group the buttons for layout purposes, you also need to add them to a container such as a JPanel.

If you look again at Figure 11.12, you will note that the appearance of the radio buttons is different from that of checkboxes in Figure 11.13. Checkboxes are square and contain a checkmark when selected. Radio buttons are round and

contain a dot when selected.

The event notification mechanism for radio buttons is the same as for any other buttons. When the user checks a radio button, the button generates an action event. In our example program, we define an action listener that sets the font size to a particular value:

```
ActionListener listener = _ -> label.setFont(new Font("Serif", Font.PL
```

Compare this listener setup to that of the checkbox example. Each radio button gets a different listener object. Each listener object knows exactly what it needs to do—set the font size to a particular value. With checkboxes, we used a different approach: Both checkboxes have the same action listener that calls a method looking at the current state of both checkboxes.

Could we follow the same approach here? We could have a single listener that computes the size as follows:

```
if (smallButton.isSelected()) size = 8;
else if (mediumButton.isSelected()) size = 12;
. . .
```

However, we prefer to use separate action listener objects because they tie the size values more closely to the buttons.

> If you have a group of radio buttons, you know that only one of them is selected. It would be nice to be able to quickly find out which, without having to query all the buttons in the group. The ButtonGroup object controls all buttons, so it would be convenient if this object could give us a reference to the selected button. Indeed, the ButtonGroup class has a getSelection method, but that method doesn't return the radio button that is selected. Instead, it returns a ButtonModel reference to the model attached to the button. Unfortunately, none of the ButtonModel methods are very helpful. The ButtonModel interface inherits a method getSelectedObjects from the ItemSelectable interface that, rather uselessly, returns

null. The getActionCommand method looks promising because the “action command” of a radio button is its text label. But the action command of its model is null. Only if you explicitly set the action commands of all radio buttons with the setActionCommand method do the action command values of the models also get set. Then you can retrieve the action command of the currently selected button with buttonGroup.getSelection().getActionCommand().

Listing 11.3 is the complete program for font size selection that puts a set of radio buttons to work.

Listing 11.3 **v2ch11/radioButton/RadioButtonFrame.java**

```
package v2ch11.radioButton;

import module java.desktop;

/**
 * A frame with a sample text label and radio buttons for selecting font sizes.
 */
public class RadioButtonFrame extends JFrame {
   private JPanel buttonPanel;
   private ButtonGroup group;
   private JLabel label;
   private static final int DEFAULT_SIZE = 36;

   public RadioButtonFrame() {
      // add the sample text label

      label = new JLabel("The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
      add(label, BorderLayout.CENTER);

      // add the radio buttons

      buttonPanel = new JPanel();
      group = new ButtonGroup();

      addRadioButton("Small", 8);
```

```
        addRadioButton("Medium", 12);
        addRadioButton("Large", 18);
        addRadioButton("Extra large", 36);

        add(buttonPanel, BorderLayout.SOUTH);
        pack();
    }

    /**
     * Adds a radio button that sets the font size of the sample text.
     * @param name the string to appear on the button
     * @param size the font size that this button sets
     */
    public void addRadioButton(String name, int size) {
        boolean selected = size == DEFAULT_SIZE;
        var button = new JRadioButton(name, selected);
        group.add(button);
        buttonPanel.add(button);

        // this listener sets the label font size

        ActionListener listener = _ -> label.setFont(new Font("Serif", Font.PLAIN, s

        button.addActionListener(listener);
    }
}
```

javax.swing.JRadioButton 1.2

- JRadioButton(String label, Icon icon)
  constructs a radio button that is initially unselected.
- JRadioButton(String label, boolean state)
  constructs a radio button with the given label and initial state.

javax.swing.ButtonGroup 1.2

- void add(AbstractButton b)
  adds the button to the group.
- ButtonModel getSelection()
  returns the button model of the selected button.

javax.swing.ButtonModel1.2

- String getActionCommand()
  returns the action command for this button model.

javax.swing.AbstractButton1.2

- void setActionCommand(String s)
  sets the action command for this button and its model.

11.4.3. Borders

If you have multiple groups of radio buttons in a window, you will want to visually indicate which buttons are grouped. Swing provides a set of useful borders for this purpose. You can apply a border to any component that extends JComponent. The most common usage is to place a border around a panel and fill that panel with other user interface elements, such as radio buttons.

You can choose from quite a few borders, but you need to follow the same steps for all of them.

1. Call a static method of the BorderFactory to create a border. You can choose among the following styles (see Figure 11.14):
   - Lowered bevel
   - Raised bevel
   - Etched
   - Line
   - Matte
   - Empty (just to create some blank space around the component)

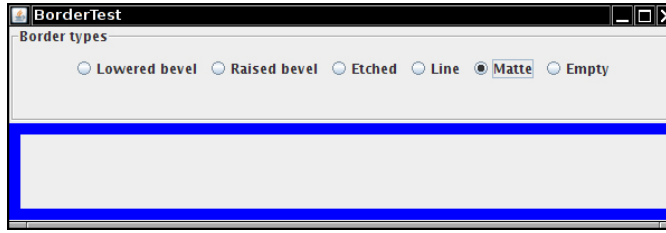


Figure 11.14: Testing border types

8. If you like, add a title to your border by passing your border to BorderFactory.createTitledBorder.
9. If you really want to go all out, combine several borders with a call to BorderFactory.createCompoundBorder.
10. Add the resulting border to your component by calling the setBorder method of the JComponent class.

For example, here is how you add an etched border with a title to a panel:

```
Border etched = BorderFactory.createEtchedBorder();
Border titled = BorderFactory.createTitledBorder(etched, "A Title");
panel.setBorder(titled);
```

Different borders have different options for setting border widths and colors; see the API notes for details. True border enthusiasts will appreciate that there is also a SoftBevelBorder class for beveled borders with softened corners and that a LineBorder can have rounded corners as well. You can construct these borders only by using one of the class constructors—there is no BorderFactory method for them.

javax.swing.BorderFactory 1.2

- static Border createLineBorder(Color color)
- static Border createLineBorder(Color color, int thickness)
  create a simple line border.
- static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)

- static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)
  create a thick border that is filled with a color or a repeating icon.
- static Border createEmptyBorder()
- static Border createEmptyBorder(int top, int left, int bottom, int right)
  create an empty border.
- static Border createEtchedBorder()
- static Border createEtchedBorder(Color highlight, Color shadow)
- static Border createEtchedBorder(int type)
- static Border createEtchedBorder(int type, Color highlight, Color shadow)
  create a line border with a 3D effect. The type parameter is one of EtchedBorder.RAISED, EtchedBorder.LOWERED.
- static Border createBevelBorder(int type)
- static Border createBevelBorder(int type, Color highlight, Color shadow)
- static Border createLoweredBevelBorder()
- static Border createRaisedBevelBorder()
  create a border that gives the effect of a lowered or raised surface. The type parameter is one of BevelBorder.RAISED, BevelBorder.LOWERED.
- static TitledBorder createTitledBorder(String title)
- static TitledBorder createTitledBorder(Border border)
- static TitledBorder createTitledBorder(Border border, String title)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font, Color color)
  create a titled border with the specified properties. The justification parameter is one of the TitledBorder constants LEFT, CENTER, RIGHT, LEADING, TRAILING, or DEFAULT_JUSTIFICATION (left), and position is one of ABOVE_TOP, TOP, BELOW_TOP, ABOVE_BOTTOM,

BOTTOM, BELOW_BOTTOM, or DEFAULT_POSITION (top).

- static CompoundBorder createCompoundBorder(Border outsideBorder, Border insideBorder)
  combines two borders into a new border.

javax.swing.border.SoftBevelBorder 1.2

- SoftBevelBorder(int type)
- SoftBevelBorder(int type, Color highlight, Color shadow)
  create a bevel border with softened corners. The type parameter is one of SoftBevelBorder.RAISED, SoftBevelBorder.LOWERED.

javax.swing.border.LineBorder 1.2

- public LineBorder(Color color, int thickness, boolean roundedCorners)
  creates a line border with the given color and thickness. If roundedCorners is true, the border has rounded corners.

javax.swing.JComponent 1.2

- void setBorder(Border border)
  sets the border of this component.

11.4.4. Combo Boxes

If you have more than a handful of alternatives, radio buttons are not a good choice because they take up too much screen space. Instead, you can use a combo box. When the user clicks on this component, a list of choices drops down, and the user can then select one of them (see Figure 11.15).

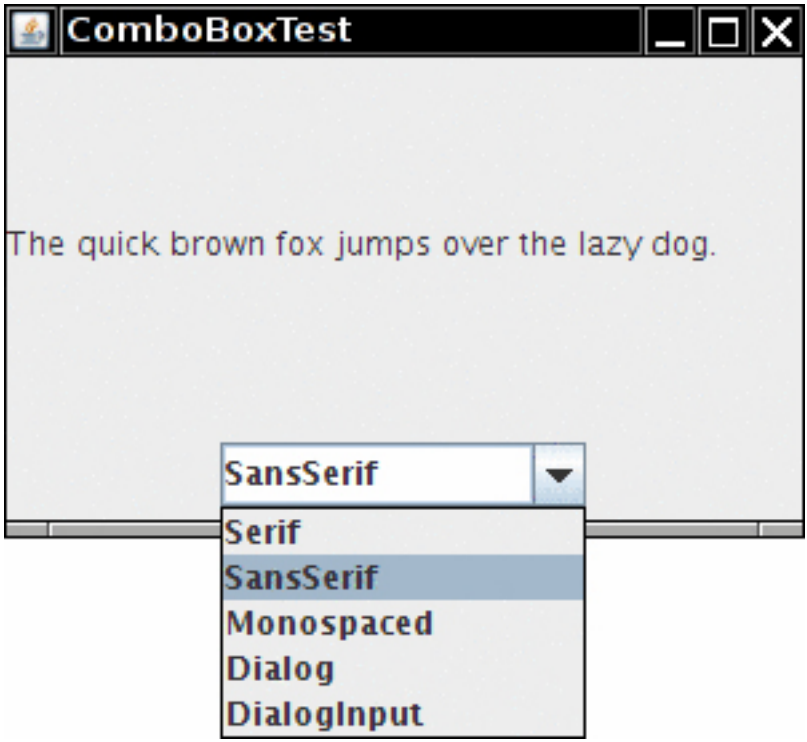


Figure 11.15: A combo box

If the drop-down list box is set to be editable, you can edit the current selection as if it were a text field. For that reason, this component is called a combo box—it combines the flexibility of a text field with a set of predefined choices. The JComboBox class provides a combo box component.

As of Java 7, the JComboBox class is a generic class. For example, a JComboBox<String> holds objects of type String, and a JComboBox<Integer> holds integers.

Call the setEditable method to make the combo box editable. Note that editing affects only the selected item. It does not change the list of choices in any way.

You can obtain the current selection, which may have been edited if the combo box is editable, by calling the getSelectedItem method. However, for an editable combo box,

that item may have any type, depending on the editor that takes the user edits and turns the result into an object. (See Chapter 12 for a discussion of editors.) If your combo box isn't editable, you are better off calling

```
combo.getItemAt(combo.getSelectedIndex())
```

which gives you the selected item with the correct type.

In the example program, the user can choose a font style from a list of styles (Serif, SansSerif, Monospaced, etc.). The user can also type in another font.

Add the choice items with the addItem method. In our program, addItem is called only in the constructor, but you can call it any time.

```
var faceCombo = new JComboBox<String>();
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
. . .
```

This method adds the string to the end of the list. You can add new items anywhere in the list with the insertItemAt method:

```
faceCombo.insertItemAt("Monospaced", 0); // add at the beginning
```

You can add items of any type—the combo box invokes each item's toString method to display it.

If you need to remove items at runtime, use the removeItem or removeItemAt method, depending on whether you supply the item to be removed or its position.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // remove first item
```

The removeAllItems method removes all items at once.

> If you need to add a large number of items to a combo box, the addItem method will perform poorly. Instead,

construct a DefaultComboBoxModel, populate it by calling addElement, and then call the setModel method of the JComboBox class.

When the user selects an item from a combo box, the combo box generates an action event. To find out which item was selected, call getSource on the event parameter to get a reference to the combo box that sent the event. Then call the getSelectedItem method to retrieve the currently selected item. You will need to cast the returned value to the appropriate type, usually String.

```
ActionListener listener = _ ->
   label.setFont(new Font(
      faceCombo.getItemAt(faceCombo.getSelectedIndex()),
      Font.PLAIN,
      DEFAULT_SIZE));
```

Listing 11.4 shows the complete program.

Listing 11.4 **v2ch11/comboBox/ComboBoxFrame.java**

```
package v2ch11.comboBox;

import module java.desktop;

/**
 * A frame with a sample text label and a combo box for selecting font faces.
 */
public class ComboBoxFrame extends JFrame {
   private JComboBox<String> faceCombo;
   private JLabel label;
   private static final int DEFAULT_SIZE = 24;

   public ComboBoxFrame() {
      // add the sample text label

      label = new JLabel("The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
      add(label, BorderLayout.CENTER);
```

```
        // make a combo box and add face names

        faceCombo = new JComboBox<>();
        faceCombo.addItem("Serif");
        faceCombo.addItem("SansSerif");
        faceCombo.addItem("Monospaced");
        faceCombo.addItem("Dialog");
        faceCombo.addItem("DialogInput");

        // the combo box listener changes the label font to the selected face name

        faceCombo.addActionListener(_ -> label.setFont(new Font(
              faceCombo.getItemAt(faceCombo.getSelectedIndex()), Font.PLAIN, DE

        // add combo box to a panel at the frame's southern border

        var comboPanel = new JPanel();
        comboPanel.add(faceCombo);
        add(comboPanel, BorderLayout.SOUTH);
        pack();
    }
}
```

javax.swing.JComboBox 1.2

- boolean isEditable()
- void setEditable(boolean b)
  get or set the editable property of this combo box.
- void addItem(Object item)
  adds an item to the item list.
- void insertItemAt(Object item, int index)
  inserts an item into the item list at a given index.
- void removeItem(Object item)
  removes an item from the item list.
- void removeItemAt(int index)
  removes the item at an index.
- void removeAllItems()
  removes all items from the item list.
- Object getSelectedItem()
  returns the currently selected item.

11.4.5. Sliders

Combo boxes let users choose from a discrete set of values. Sliders offer a choice from a continuum of values—for example, any number between 1 and 100.

The most common way of constructing a slider is as follows:

```
var slider = new JSlider(min, max, initialValue);
```

If you omit the minimum, maximum, and initial values, they are initialized with 0, 100, and 50, respectively.

Or if you want the slider to be vertical, use the following constructor call:

```
var slider = new JSlider(SwingConstants.VERTICAL, min, max, initial
```

These constructors create a plain slider, such as the top slider in Figure 11.16. You will see presently how to add decorations to a slider.

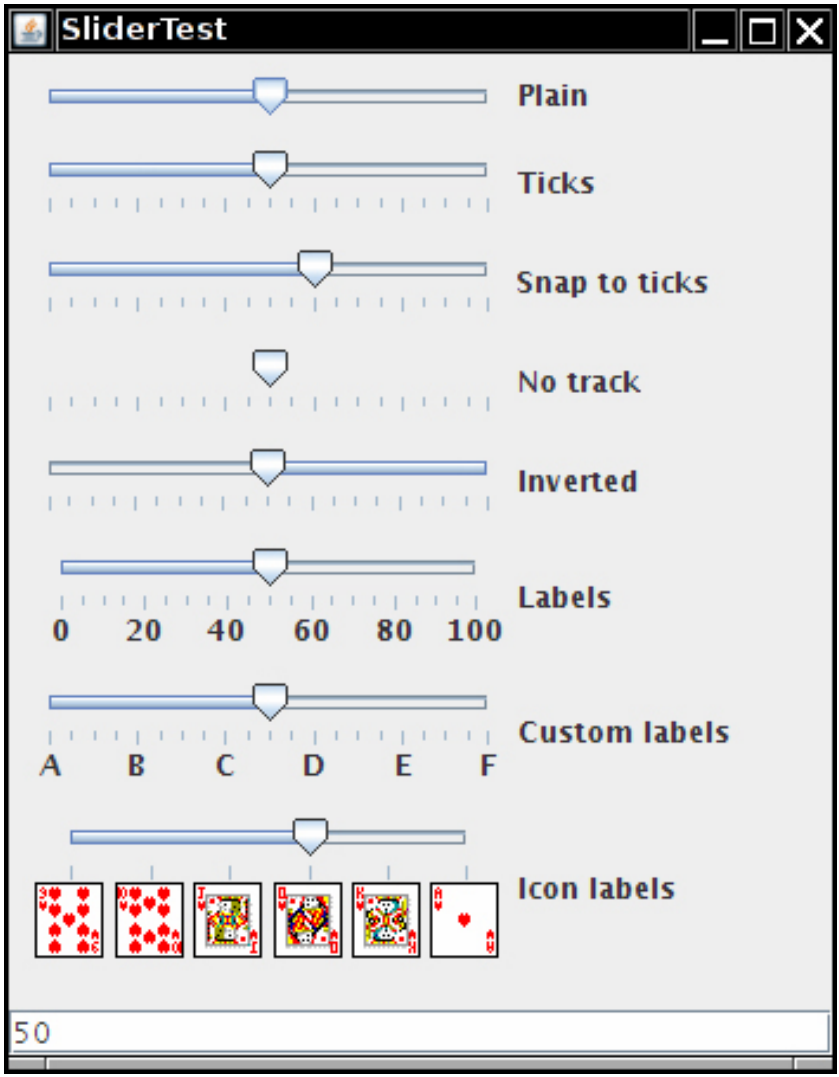


Figure 11.16: Sliders

As the user slides the slider bar, the value of the slider moves between the minimum and the maximum values. When the value changes, a ChangeEvent is sent to all change listeners. To be notified of the change, call the addChangeListener method and install an object that implements the functional ChangeListener interface. In the callback, retrieve the slider value:

```
ChangeListener listener = _ -> {
   JSlider slider = (JSlider) event.getSource();
   int value = slider.getValue();
   . . .
};
```

You can embellish the slider by showing ticks. For example, in the sample program, the second slider uses the following settings:

```
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
```

The slider is decorated with large tick marks every 20 units and small tick marks every 5 units. The units refer to slider values, not pixels.

These instructions only set the units for the tick marks. To actually have the tick marks appear, call

```
slider.setPaintTicks(true);
```

The major and minor tick marks are independent. For example, you can set major tick marks every 20 units and minor tick marks every 7 units, but that will give you a very messy scale.

You can force the slider to snap to ticks. Whenever the user has finished dragging a slider in snap mode, it is immediately moved to the closest tick. You activate this mode with the call

```
slider.setSnapToTicks(true);
```

> The "snap to ticks" behavior doesn't work as well as you might imagine. Until the slider has actually snapped, the change listener still reports slider values that don't correspond to ticks. And if you click next to the slider—an action that normally advances the slider a bit in the direction of the click—a slider with "snap to ticks" does not move to the next tick.

You can display tick mark labels for the major tick marks by calling

```
slider.setPaintLabels(true);
```

For example, with a slider ranging from 0 to 100 and major tick spacing of 20, the ticks are labeled 0, 20, 40, 60, 80, and 100.

You can also supply other tick mark labels, such as strings or icons (see Figure 11.16). The process is a bit convoluted. You need to fill a hash table with keys of type Integer and values of type Component. You then call the setLabelTable method. The components are placed under the tick marks. Usually, JLabel objects are used. Here is how you can label ticks as A, B, C, D, E, and F:

```
var labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));
. . .
labelTable.put(100, new JLabel("F"));
slider.setLabelTable(labelTable);
```

Listing 11.5 also shows a slider with icons as tick labels.

If your tick marks or labels don't show, double-check that you called setPaintTicks(true) and setPaintLabels(true).

The fourth slider in Figure 11.16 has no track. To suppress the "track" in which the slider moves, call

```
slider.setPaintTrack(false);
```

The fifth slider has its direction reversed by a call to

```
slider.setInverted(true);
```

The example program in Listing 11.5 shows all these visual effects with a collection of sliders. Each slider has a change event listener installed that places the current slider value into the text field at the bottom of the frame.

Listing 11.5 **v2ch11/slider/SliderFrame.java**

```
package v2ch11.slider;

import module java.base;
import module java.desktop;

/**
 * A frame with many sliders and a text field to show slider values.
 */
public class SliderFrame extends JFrame {
   private JPanel sliderPanel;
   private JTextField textField;
   private ChangeListener listener;

   public SliderFrame() {
      sliderPanel = new JPanel();
      sliderPanel.setLayout(new GridBagLayout());

      // common listener for all sliders
      listener = event -> {
         // update text field when the slider value changes
         JSlider source = (JSlider) event.getSource();
         textField.setText("" + source.getValue());
      };

      // add a plain slider

      var slider = new JSlider();
      addSlider(slider, "Plain");

      // add a slider with major and minor ticks

      slider = new JSlider();
      slider.setPaintTicks(true);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(5);
      addSlider(slider, "Ticks");

      // add a slider that snaps to ticks

      slider = new JSlider();
```

```
slider.setPaintTicks(true);
slider.setSnapToTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
addSlider(slider, "Snap to ticks");

// add a slider with no track

slider = new JSlider();
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
slider.setPaintTrack(false);
addSlider(slider, "No track");

// add an inverted slider

slider = new JSlider();
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
slider.setInverted(true);
addSlider(slider, "Inverted");

// add a slider with numeric labels

slider = new JSlider();
slider.setPaintTicks(true);
slider.setPaintLabels(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
addSlider(slider, "Labels");

// add a slider with alphabetic labels

slider = new JSlider();
slider.setPaintLabels(true);
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);

var labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));
```

```
    labelTable.put(40, new JLabel("C"));
    labelTable.put(60, new JLabel("D"));
    labelTable.put(80, new JLabel("E"));
    labelTable.put(100, new JLabel("F"));

    slider.setLabelTable(labelTable);
    addSlider(slider, "Custom labels");

    // add a slider with icon labels

    slider = new JSlider();
    slider.setPaintTicks(true);
    slider.setPaintLabels(true);
    slider.setSnapToTicks(true);
    slider.setMajorTickSpacing(20);
    slider.setMinorTickSpacing(20);

    labelTable = new Hashtable<>();

    // add card images

    labelTable.put(0, new JLabel(new ImageIcon("images/nine.gif")));
    labelTable.put(20, new JLabel(new ImageIcon("images/ten.gif")));
    labelTable.put(40, new JLabel(new ImageIcon("images/jack.gif")));
    labelTable.put(60, new JLabel(new ImageIcon("images/queen.gif")));
    labelTable.put(80, new JLabel(new ImageIcon("images/king.gif")));
    labelTable.put(100, new JLabel(new ImageIcon("images/ace.gif")));

    slider.setLabelTable(labelTable);
    addSlider(slider, "Icon labels");

    // add the text field that displays the slider value

    textField = new JTextField();
    add(sliderPanel, BorderLayout.CENTER);
    add(textField, BorderLayout.SOUTH);
    pack();
}

/**
 * Adds a slider to the slider panel and hooks up the listener.
 * @param slider the slider
 * @param description the slider description
 */
```

```
   public void addSlider(JSlider slider, String description) {
      slider.addChangeListener(listener);
      var panel = new JPanel();
      panel.add(slider);
      panel.add(new JLabel(description));
      panel.setAlignmentX(Component.LEFT_ALIGNMENT);
      var gbc = new GridBagConstraints();
      gbc.gridy = sliderPanel.getComponentCount();
      gbc.anchor = GridBagConstraints.WEST;
      sliderPanel.add(panel, gbc);
   }
}
```

javax.swing.JSlider 1.2

- JSlider()
- JSlider(int direction)
- JSlider(int min, int max)
- JSlider(int min, int max, int initialValue)
- JSlider(int direction, int min, int max, int initialValue)
  construct a slider with the given direction and minimum, maximum, and initial values. The direction parameter is one of SwingConstants.HORIZONTAL or SwingConstants.VERTICAL. The default is horizontal. Defaults for the minimum, initial, and maximum are 0, 50, and 100.
- void setPaintTicks(boolean b)
  displays ticks if b is true.
- void setMajorTickSpacing(int units)
- void setMinorTickSpacing(int units)
  set major or minor ticks at multiples of the given slider units.
- void setPaintLabels(boolean b)
  displays tick labels if b is true.
- void setLabelTable(Dictionary table)
  sets the components to use for the tick labels. Each key/value pair in the table has the form Integer.valueOf(value)/component.
- void setSnapToTicks(boolean b)
  If b is true, then the slider snaps to the closest tick after each adjustment.

- void setPaintTrack(boolean b)
  If b is true, a track is displayed in which the slider runs.

11.5. Menus

This chapter started by introducing the most common components that you might want to place into a window, such as various kinds of buttons, text fields, and combo boxes. Swing also supports another type of user interface element—pull-down menus that are familiar from GUI applications.

A menu bar at the top of a window contains the names of the pull-down menus. Clicking on a name opens the menu containing menu items and submenus. When the user clicks on a menu item, all menus are closed and a message is sent to the program. Figure 11.17 shows a typical menu with a submenu.

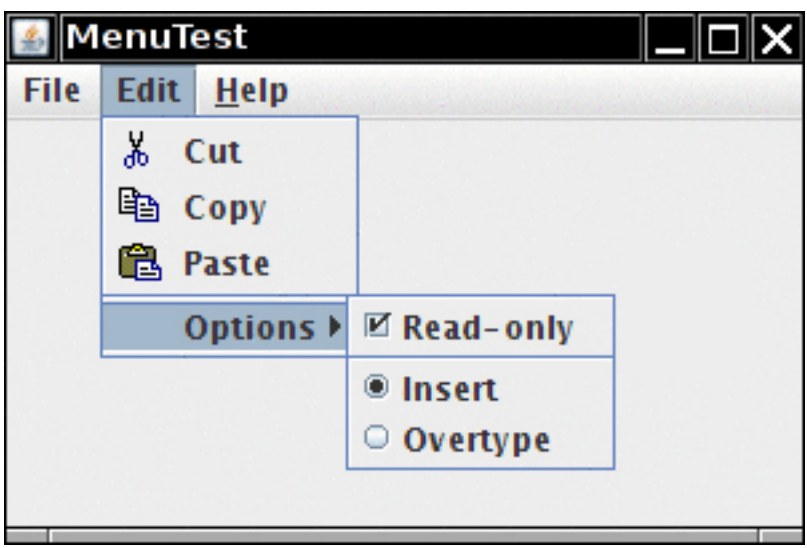


Figure 11.17: A menu with a submenu

11.5.1. Menu Building

Building menus is straightforward. First, create a menu bar:

```
var menuBar = new JMenuBar();
```

A menu bar is just a component that you can add anywhere you like. Normally, you want it to appear at the top of a frame. You can add it there with the setJMenuBar method:

```
frame.setJMenuBar(menuBar);
```

For each menu, you create a menu object:

```
var editMenu = new JMenu("Edit");
```

Add the top-level menus to the menu bar:

```
menuBar.add(editMenu);
```

Add menu items, separators, and submenus to the menu object:

```
var pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // a submenu
editMenu.add(optionsMenu);
```

You can see separators in Figure 11.17 below the Paste and Read-only menu items.

When the user selects a menu item, an action event is triggered. You need to install an action listener for each menu item:

```
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
```

The method JMenu.add(String s) conveniently adds a menu item to the end of a menu. For example:

```
editMenu.add("Paste");
```

The add method returns the created menu item, so you can

capture it and add the listener, as follows:

```
JMenuItem pasteItem = editMenu.add("Paste");
pasteItem.addActionListener(listener);
```

It often happens that menu items trigger commands that can also be activated through other user interface elements such as toolbar buttons. In Chapter 10, you saw how to specify commands through Action objects. You define a class that implements the Action interface, usually by extending the AbstractAction convenience class, specify the menu item label in the constructor of the AbstractAction object, and override the actionPerformed method to hold the menu action handler. For example:

```
var exitAction = new AbstractAction("Exit") { // menu item text goes h
   public void actionPerformed(ActionEvent event) {
      // action code goes here
      System.exit(0);
   }
};
```

You can then add the action to the menu:

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

This command adds a menu item to the menu, using the action name. The action object becomes its listener. This is just a convenient shortcut for

```
var exitItem = new JMenuItem(exitAction);
fileMenu.add(exitItem);
```

javax.swing.JMenu 1.2

- JMenu(String label)
  constructs a menu with the given label.
- JMenuItem add(JMenuItem item)
  adds a menu item (or a menu).
- JMenuItem add(String label)

adds a menu item with the given label to this menu and returns the item.

- JMenuItem add(Action a)
  adds a menu item with the given action to this menu and returns the item.
- void addSeparator()
  adds a separator line to the menu.
- JMenuItem insert(JMenuItem menu, int index)
  adds a new menu item (or submenu) to the menu at a specific index.
- JMenuItem insert(Action a, int index)
  adds a new menu item with the given action at a specific index.
- void insertSeparator(int index)
  adds a separator to the menu.
- void remove(int index)
- void remove(JMenuItem item)
  remove a specific item from the menu.

javax.swing.JMenuItem1.2

- JMenuItem(String label)
  constructs a menu item with a given label.
- JMenuItem(Action a)1.3
  constructs a menu item for the given action.

javax.swing.AbstractButton1.2

- void setAction(Action a)1.3
  sets the action for this button or menu item.

javax.swing.JFrame1.2

- void setJMenuBar(JMenuBar menubar)
  sets the menu bar for this frame.

11.5.2. Icons in Menu Items

Menu items are very similar to buttons. In fact, the JMenuItem class extends the AbstractButton class. Just like buttons, menus can have just a text label, just an icon, or both. You can specify the icon with the JMenuItem(String, Icon) or JMenuItem(Icon) constructor, or you can set it with the setIcon method that the JMenuItem class inherits from the AbstractButton class. Here is an example:

```
var cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));
```

In Figure 11.17, you can see icons next to several menu items. By default, the menu item text is placed to the right of the icon. If you prefer the text to be placed on the left, call the setHorizontalTextPosition method that the JMenuItem class inherits from the AbstractButton class. For example, the call

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

moves the menu item text to the left of the icon.

You can also add an icon to an action:

```
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
```

Whenever you construct a menu item out of an action, the Action.NAME value becomes the text of the menu item and the Action.SMALL_ICON value becomes the icon.

Alternatively, you can set the icon in the AbstractAction constructor:

```
cutAction = new AbstractAction("Cut", new ImageIcon("cut.gif")) {
      public void actionPerformed(ActionEvent event) {
         . . .
      }
   };
```

javax.swing.JMenuItem1.2

- JMenuItem(String label, Icon icon)
  constructs a menu item with the given label and icon.

javax.swing.AbstractButton1.2

- void setHorizontalTextPosition(int pos)
  sets the horizontal position of the text relative to the icon. The pos parameter is SwingConstants.RIGHT (text is to the right of icon) or SwingConstants.LEFT.

javax.swing.AbstractAction1.2

- AbstractAction(String name, Icon smallIcon)
  constructs an abstract action with the given name and icon.

11.5.3. Checkbox and Radio Button Menu Items

Checkbox and radio button menu items display a checkbox or radio button next to the name (see Figure 11.17). When the user selects the menu item, the item automatically toggles between checked and unchecked.

Apart from the button decoration, treat these menu items just as you would any others. For example, here is how you create a checkbox menu item:

```
var readonlyItem = new JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

The radio button menu items work just like regular radio buttons. You must add them to a button group. When one of the buttons in a group is selected, all others are automatically deselected.

```
var group = new ButtonGroup();
```

```
var insertItem = new JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
var overtypeItem = new JRadioButtonMenuItem("Overtype");
group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

With these menu items, you don't necessarily want to be notified when the user selects the item. Instead, you can simply use the isSelected method to test the current state of the menu item. (Of course, that means you should keep a reference to the menu item stored in an instance field.) Use the setSelected method to set the state.

javax.swing.JCheckBoxMenuItem 1.2

- JCheckBoxMenuItem(String label)
  constructs the checkbox menu item with the given label.
- JCheckBoxMenuItem(String label, boolean state)
  constructs the checkbox menu item with the given label and the given initial state (true is checked).

javax.swing.JRadioButtonMenuItem 1.2

- JRadioButtonMenuItem(String label)
  constructs the radio button menu item with the given label.
- JRadioButtonMenuItem(String label, boolean state)
  constructs the radio button menu item with the given label and the given initial state (true is checked).

javax.swing.AbstractButton 1.2

- boolean isSelected()
- void setSelected(boolean state)
  get or set the selection state of this item (true is checked).

11.5.4. Pop-Up Menus

A pop-up menu is a menu that is not attached to a menu bar but floats somewhere (see Figure 11.18).

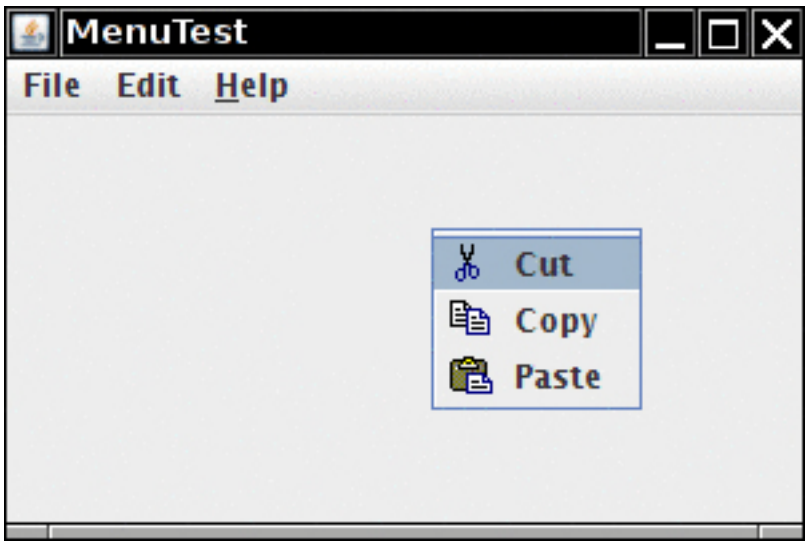


Figure 11.18: A pop-up menu

Create a pop-up menu just as you create a regular menu, except that a pop-up menu has no title.

```
var popup = new JPopupMenu();
```

Then, add your menu items as usual:

```
var item = new JMenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

Unlike the regular menu bar that is always shown at the top of the frame, you must explicitly display a pop-up menu by using the show method. Specify the parent component and the location of the pop-up, using the coordinate system of the parent. For example:

```
popup.show(panel, x, y);
```

Usually, you want to pop up a menu when the user clicks a particular mouse button—the so-called pop-up trigger. In Windows and Linux, the pop-up trigger is the nonprimary (usually, the right) mouse button. To pop up a menu when the user clicks on a component, using the pop-up trigger, simply call the method

```
component.setComponentPopupMenu(popup);
```

Very occasionally, you may place a component inside another component that has a pop-up menu. The child component can inherit the parent component's pop-up menu by calling

```
child.setInheritsPopupMenu(true);
```

javax.swing.JPopupMenu 1.2

- void show(Component c, int x, int y)
  shows the pop-up menu over the component c with the top left corner at (x, y) (in the coordinate space of c).
- boolean isPopupTrigger(MouseEvent event) 1.3
  returns true if the mouse event is the pop-up menu trigger.

java.awt.event.MouseEvent 1.1

- boolean isPopupTrigger()
  returns true if this mouse event is the pop-up menu trigger.

javax.swing.JComponent 1.2

- JPopupMenu getComponentPopupMenu() 5.0
- void setComponentPopupMenu(JPopupMenu popup) 5.0
  get or set the pop-up menu for this component.
- boolean getInheritsPopupMenu() 5.0

- void setInheritsPopupMenu(boolean b)5.0
  get or set the inheritsPopupMenu property. If the property is set and this component's pop-up menu is null, it uses its parent's pop-up menu.

11.5.5. Keyboard Mnemonics and Accelerators

It is a real convenience for the experienced user to select menu items by keyboard mnemonics. You can create a keyboard mnemonic for a menu item by specifying a mnemonic letter in the menu item constructor:

```
var aboutItem = new JMenuItem("About", 'A');
```

The keyboard mnemonic is displayed automatically in the menu, with the mnemonic letter underlined (see Figure 11.19). For example, in the item defined in the last example, the label will be displayed as "About" with an underlined letter 'A'. When the menu is displayed, the user just needs to press the A key, and the menu item is selected. (If the mnemonic letter is not part of the menu string, then typing it still selects the item, but the mnemonic is not displayed in the menu. Naturally, such invisible mnemonics are of dubious utility.)

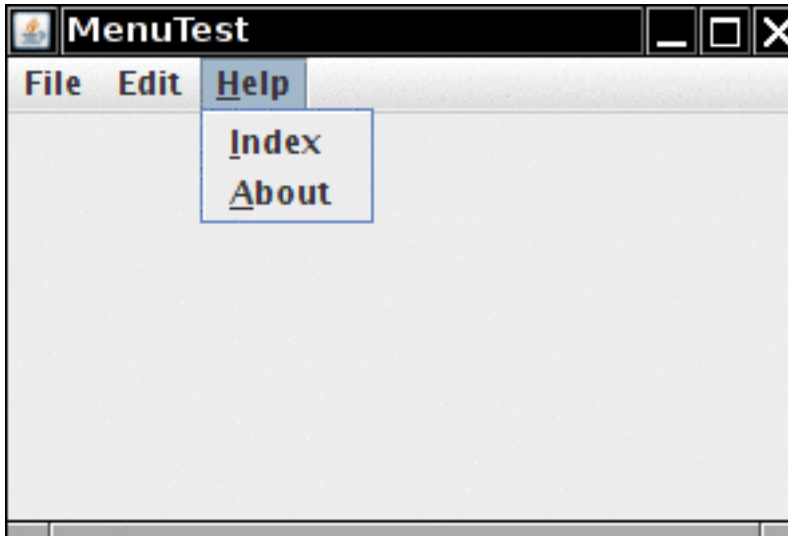


Figure 11.19: Keyboard mnemonics

Sometimes, you don't want to underline the first letter of the menu item that matches the mnemonic. For example, if you have a mnemonic "A" for the menu item "Save As," then it makes more sense to underline the second "A" (Save As). You can specify which character you want to have underlined by calling the setDisplayedMnemonicIndex method.

If you have an Action object, you can add the mnemonic as the value of the Action.MNEMONIC_KEY key, as follows:

```
aboutAction.putValue(Action.MNEMONIC_KEY, Integer.valueOf('A'));
```

You can supply a mnemonic letter only in the constructor of a menu item, not in the constructor for a menu. To attach a mnemonic to a menu, call the setMnemonic method:

```
var helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');
```

To select a top-level menu from the menu bar, press the Alt key together with the mnemonic letter. For example, press Alt+H to select the Help menu from the menu bar.

Keyboard mnemonics let you select a submenu or menu item from the currently open menu. In contrast, accelerators are keyboard shortcuts that let you select menu items without ever opening a menu. For example, many programs attach the accelerators Ctrl+O and Ctrl+S to the Open and Save items in the File menu. Use the setAccelerator method to attach an accelerator key to a menu item. The setAccelerator method takes an object of type Keystroke. For example, the following call attaches the accelerator Ctrl+O to the openItem menu item:

```
openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
```

Typing the accelerator key combination automatically selects the menu option and fires an action event, as if the user had selected the menu option manually.

You can attach accelerators only to menu items, not to menus. Accelerator keys don't actually open the menu. Instead, they directly fire the action event associated with a menu.

Conceptually, adding an accelerator to a menu item is similar to the technique of adding an accelerator to a Swing component. However, when the accelerator is added to a menu item, the key combination is automatically displayed in the menu (see Figure 11.20).

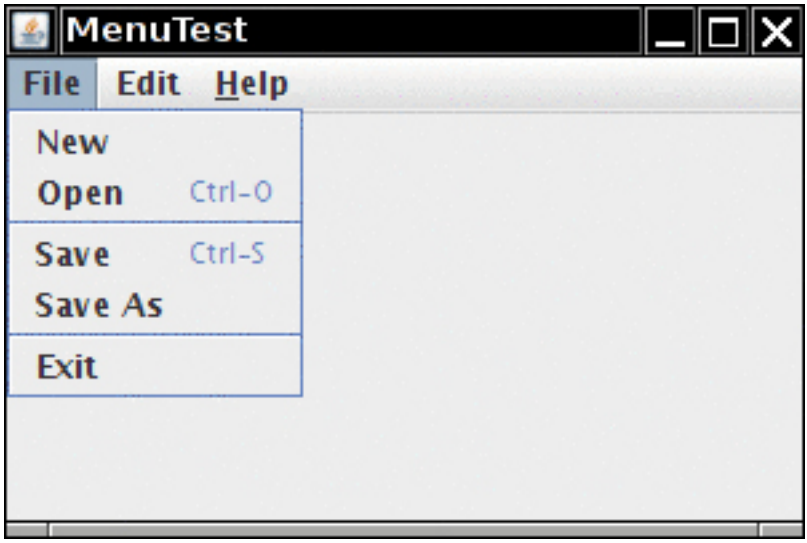


Figure 11.20: Accelerators

Under Windows, Alt+F4 closes a window. But this is not an accelerator to be programmed in Java. It is a shortcut defined by the operating system. This key combination will always trigger the WindowClosing event for the active window regardless of whether there is a Close item on the menu.

---

javax.swing.JMenuItem1.2

- JMenuItem(String label, int mnemonic)
  constructs a menu item with a given label and mnemonic.
- void setAccelerator(KeyStroke k)
  sets the keystroke k as accelerator for this menu item. The accelerator key is displayed next to the label.

javax.swing.AbstractButton1.2

- void setMnemonic(int mnemonic)
  sets the mnemonic character for the button. This character

will be underlined in the label.

- void setDisplayedMnemonicIndex(int index)1.4
  sets the index of the character to be underlined in the button text. Use this method if you don't want the first occurrence of the mnemonic character to be underlined.

11.5.6. Enabling and Disabling Menu Items

Occasionally, a particular menu item should be selected only in certain contexts. For example, when a document is opened in read-only mode, the Save menu item is not meaningful. Of course, we could remove the item from the menu with the JMenu.remove method, but users would react with some surprise to menus whose content keeps changing. Instead, it is better to deactivate the menu items that lead to temporarily inappropriate commands. A deactivated menu item is shown in gray and cannot be selected (see Figure 11.21).

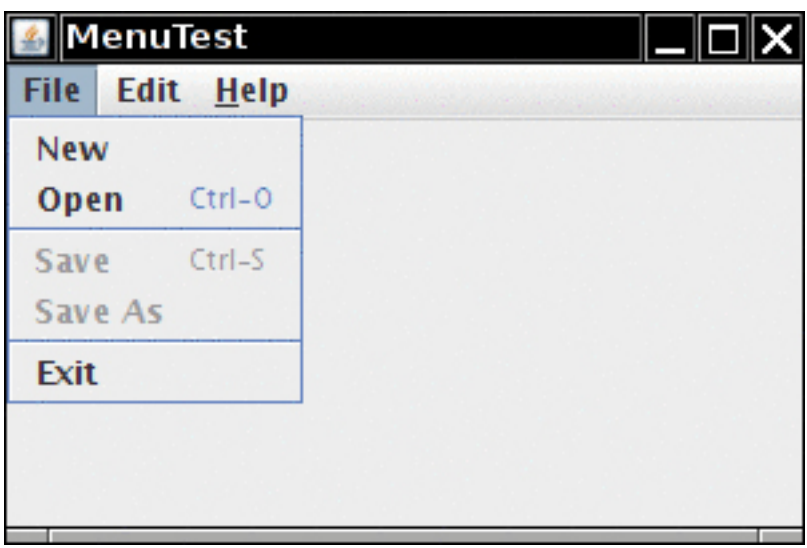


Figure 11.21: Disabled menu items

To enable or disable a menu item, use the setEnabled method:

```
saveItem.setEnabled(false);
```

There are two strategies for enabling and disabling menu items. Each time circumstances change, you can call setEnabled on the relevant menu items or actions. For example, as soon as a document has been set to read-only mode, you can locate the Save and Save As menu items and disable them. Alternatively, you can disable items just before displaying the menu. To do this, you must register a listener for the "menu selected" event. The javax.swing.event package defines a MenuListener interface with three methods:

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

The menuSelected method is called before the menu is displayed. It can therefore be used to disable or enable menu items. The following code shows how to disable the Save and Save As actions whenever the Read Only checkbox menu item is selected:

```
public void menuSelected(MenuEvent event) {
   saveAction.setEnabled(!readonlyItem.isSelected());
   saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```

Disabling menu items just before displaying the menu is a clever idea, but it does not work for menu items that also have accelerator keys. Since the menu is never opened when the accelerator key is pressed, the action is never disabled, and is still triggered by the accelerator key.

---

javax.swing.JMenuItem **1.2**

- void setEnabled(boolean b)
  enables or disables the menu item.

javax.swing.event.MenuListener1.2

- void menuSelected(MenuEvent e)
  is called when the menu has been selected, before it is opened.
- void menuDeselected(MenuEvent e)
  is called when the menu has been deselected, after it has been closed.
- void menuCanceled(MenuEvent e)
  is called when the menu has been canceled, for example, by a user clicking outside the menu.

Listing 11.6 is a sample program that generates a set of menus. It shows all the features that you saw in this section: nested menus, disabled menu items, checkbox and radio button menu items, a pop-up menu, and keyboard mnemonics and accelerators.

Listing 11.6 **v2ch11/menu/MenuFrame.java**

```
package v2ch11.menu;

import module java.desktop;

/**
 * A frame with a sample menu bar.
 */
public class MenuFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;
   private Action saveAction;
   private Action saveAsAction;
   private JCheckBoxMenuItem readonlyItem;

   /**
    * A sample action that prints the action name to System.out.
    */
   class TestAction extends AbstractAction {
```

```
    public TestAction(String name) {
       super(name);
    }

    public void actionPerformed(ActionEvent event) {
       System.out.println(getValue(Action.NAME) + " selected.");
    }
 }

 public MenuFrame() {
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    var fileMenu = new JMenu("File");
    fileMenu.add(new TestAction("New"));

    // demonstrate accelerators

    var openItem = fileMenu.add(new TestAction("Open"));
    openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));

    fileMenu.addSeparator();

    saveAction = new TestAction("Save");
    JMenuItem saveItem = fileMenu.add(saveAction);
    saveItem.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));

    saveAsAction = new TestAction("Save As");
    fileMenu.add(saveAsAction);
    fileMenu.addSeparator();

    fileMenu.add(new AbstractAction("Exit") {
       public void actionPerformed(ActionEvent event) {
          System.exit(0);
       }
    });

    // demonstrate checkbox and radio button menus

    readonlyItem = new JCheckBoxMenuItem("Read-only");
    readonlyItem.addActionListener(_ -> {
       boolean saveOk = !readonlyItem.isSelected();
       saveAction.setEnabled(saveOk);
       saveAsAction.setEnabled(saveOk);
    });
```

```
var group = new ButtonGroup();

var insertItem = new JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
var overtypeItem = new JRadioButtonMenuItem("Overtype");

group.add(insertItem);
group.add(overtypeItem);

// demonstrate icons

var cutAction = new TestAction("Cut");
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("images/cut.gif")
var copyAction = new TestAction("Copy");
copyAction.putValue(Action.SMALL_ICON, new ImageIcon("images/copy.g
var pasteAction = new TestAction("Paste");
pasteAction.putValue(Action.SMALL_ICON, new ImageIcon("images/paste

var editMenu = new JMenu("Edit");
editMenu.add(cutAction);
editMenu.add(copyAction);
editMenu.add(pasteAction);

// demonstrate nested menus

var optionMenu = new JMenu("Options");

optionMenu.add(readonlyItem);
optionMenu.addSeparator();
optionMenu.add(insertItem);
optionMenu.add(overtypeItem);

editMenu.addSeparator();
editMenu.add(optionMenu);

// demonstrate mnemonics

var helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');

var indexItem = new JMenuItem("Index");
indexItem.setMnemonic('I');
helpMenu.add(indexItem);
```

```
        // you can also add the mnemonic key to an action
        var aboutAction = new TestAction("About");
        aboutAction.putValue(Action.MNEMONIC_KEY, Integer.valueOf('A'));
        helpMenu.add(aboutAction);

        // add all top-level menus to menu bar

        var menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        menuBar.add(fileMenu);
        menuBar.add(editMenu);
        menuBar.add(helpMenu);

        // demonstrate pop-ups

        JPopupMenu popup = new JPopupMenu();
        popup.add(cutAction);
        popup.add(copyAction);
        popup.add(pasteAction);

        var panel = new JPanel();
        panel.setComponentPopupMenu(popup);
        add(panel);
    }
}
```

11.5.7. Toolbars

A toolbar is a button bar that gives quick access to the most commonly used commands in a program (see Figure 11.22).

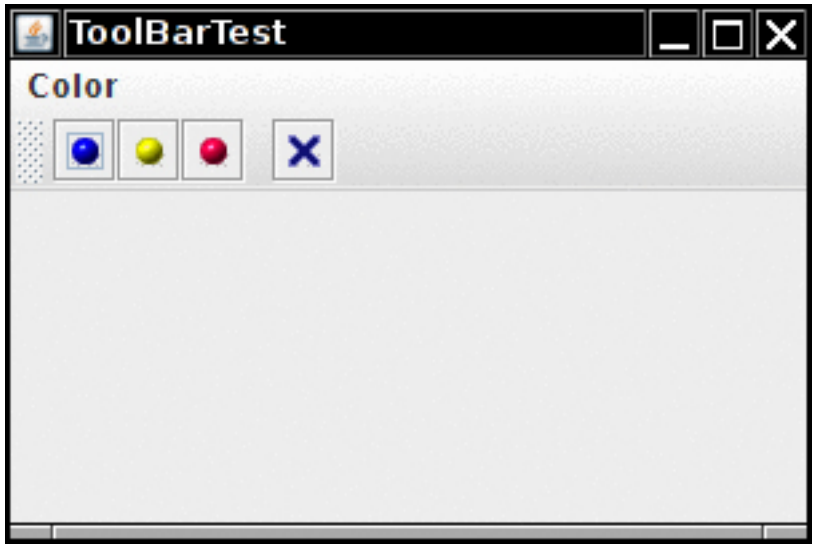


Figure 11.22: A toolbar

What makes toolbars special is that you can move them elsewhere. You can drag the toolbar to one of the four borders of the frame (see Figure 11.23). When you release the mouse button, the toolbar is dropped into the new location (see Figure 11.24).

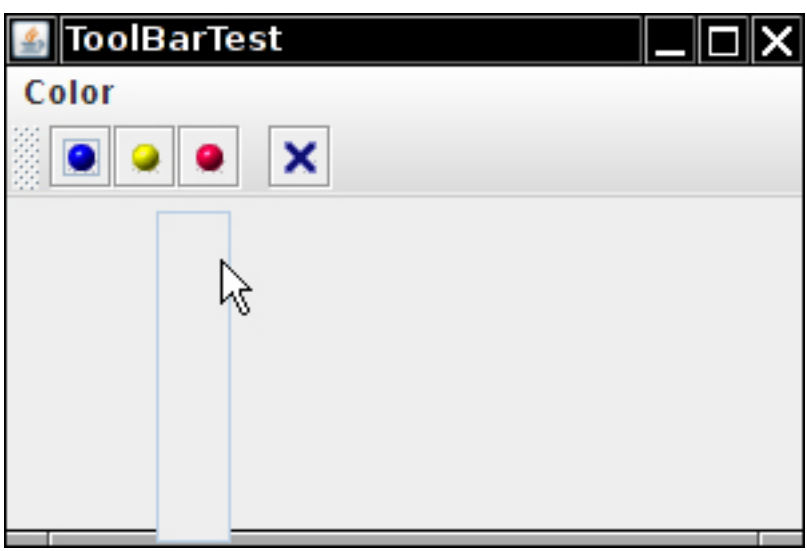


Figure 11.23: Dragging the toolbar

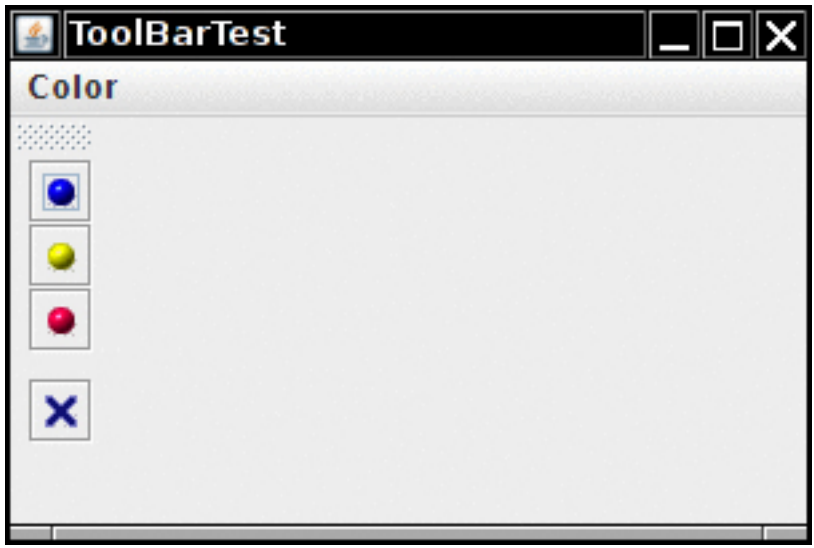


Figure 11.24: The toolbar has been dragged to another border

Toolbar dragging works if the toolbar is inside a container with a border layout, or any other layout manager that supports the North, East, South, and West constraints.

The toolbar can even be completely detached from the frame. A detached toolbar is contained in its own frame (see Figure 11.25). When you close the frame containing a detached toolbar, the toolbar jumps back into the original frame.

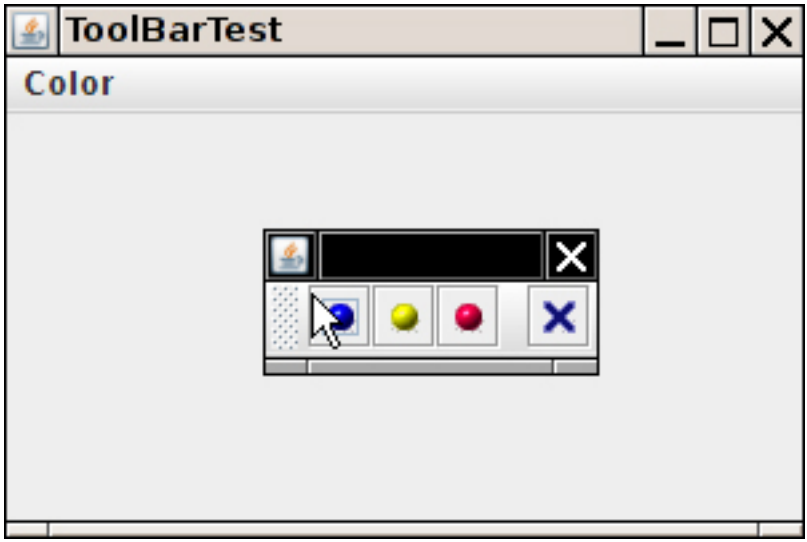


Figure 11.25: Detaching the toolbar

Toolbars are straightforward to program. Add components into the toolbar:

```
var toolbar = new JToolBar();
toolbar.add(blueButton);
```

The JToolBar class also has a method to add an Action object. Simply populate the toolbar with Action objects, like this:

```
toolbar.add(blueAction);
```

The small icon of the action is displayed in the toolbar.

You can separate groups of buttons with a separator:

```
toolbar.addSeparator();
```

For example, the toolbar in Figure 11.22 has a separator between the third and fourth buttons.

Then, add the toolbar to the frame:

```
add(toolbar, BorderLayout.NORTH);
```

You can also specify a title for the toolbar that appears when the toolbar is undocked:

```
toolbar = new JToolBar(titleString);
```

By default, toolbars are initially horizontal. To have a toolbar start out vertical, use

```
toolbar = new JToolBar(SwingConstants.VERTICAL)
```

or

```
toolbar = new JToolBar(titleString, SwingConstants.VERTICAL)
```

Buttons are the most common components inside toolbars. But there is no restriction on the components that you can add to a toolbar. For example, you can add a combo box to a toolbar.

11.5.8. Tooltips

A disadvantage of toolbars is that users are often mystified by the meanings of the tiny icons in toolbars. To solve this problem, user interface designers invented tooltips. A tooltip is activated when the cursor rests for a moment over a button. The tooltip text is displayed inside a colored rectangle. When the user moves the mouse away, the tooltip disappears. (See Figure 11.26.)

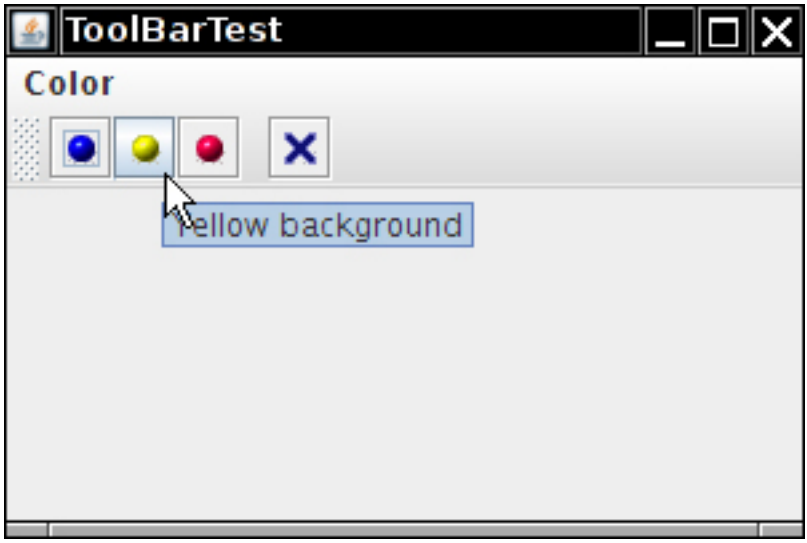


Figure 11.26: A tooltip

In Swing, you can add tooltips to any JComponent simply by calling the setToolTipText method:

```
exitButton.setToolTipText("Exit");
```

Alternatively, if you use Action objects, you associate the tooltip with the SHORT_DESCRIPTION key:

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
```

javax.swing.JToolBar 1.2

- JToolBar()
- JToolBar(String titleString)
- JToolBar(int orientation)
- JToolBar(String titleString, int orientation)
  construct a toolbar with the given title string and orientation. orientation is one of SwingConstants.HORIZONTAL (the default) or SwingConstants.VERTICAL.
- JButton add(Action a)
  constructs a new button inside the toolbar with name,

icon, short description, and action callback from the given action, and adds the button to the end of the toolbar.

- void addSeparator()
  adds a separator to the end of the toolbar.

javax.swing.JComponent1.2

- void setToolTipText(String text)
  sets the text that should be displayed as a tooltip when the mouse hovers over the component.

11.6. The Grid Bag Layout

So far we've been using only the border layout, flow layout, and grid layout for the user interface of our sample applications. For more complex tasks, this is not going to be enough.

Since Java 1.0, the AWT includes the grid bag layout that lays out components in rows and columns. The row and column sizes are flexible, and components can span multiple rows and columns. This layout manager is very flexible, but also very complex. The mere mention of the words “grid bag layout” has been known to strike fear in the hearts of Java programmers.

In an unsuccessful attempt to design a layout manager that would free programmers from the tyranny of the grid bag layout, the Swing designers came up with the box layout. According to the JDK documentation of the BoxLayout class: “Nesting multiple panels with different combinations of horizontal and vertical [sic] gives an effect similar to GridBagLayout, without the complexity.” However, as each box is laid out independently, you cannot use box layouts to arrange neighboring components both horizontally and vertically.

Java 1.4 saw yet another attempt to design a replacement for the grid bag layout—the spring layout where you use

imaginary springs to connect the components in a container. As the container is resized, the springs stretch or shrink, thereby adjusting the positions of the components. This sounds tedious and confusing, and it is. The spring layout quickly sank into obscurity.

The NetBeans IDE combines a layout tool (called “Matisse”) and a layout manager. A user interface designer uses the tool to drop components into a container and to indicate which components should line up. The tool translates the designer’s intentions into instructions for the group layout manager. This is much more convenient than writing the layout management code by hand.

In the coming sections, I will cover the grid bag layout because it is commonly used and is still the easiest mechanism for programmatically producing layout code. I will show you a strategy that makes grid bag layouts relatively painless in common situations.

11.6.1. Grid Bag Basics

The grid bag layout is the mother of all layout managers. You can think of a grid bag layout as a grid layout without the limitations. In a grid bag layout, the rows and columns can have variable sizes. You can join adjacent cells to make room for larger components. (Many word processors, as well as HTML, provide similar capabilities for tables: You can start out with a grid and then merge adjacent cells as necessary.) The components need not fill the entire cell area, and you can specify their alignment within cells.

Consider the font selector of Figure 11.27. It consists of the following components:

- Two combo boxes to specify the font face and size
- Labels for these two combo boxes
- Two checkboxes to select bold and italic
- A text area for the sample string

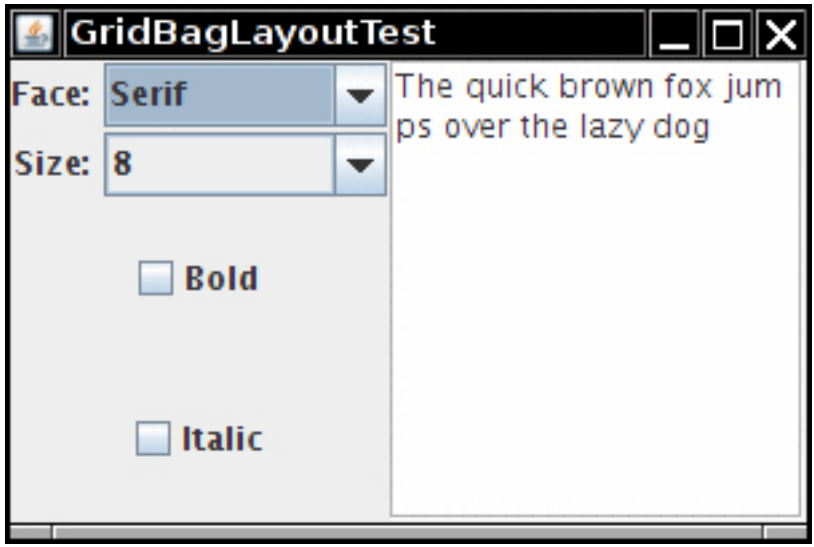


Figure 11.27: A font selector

Now, chop up the container into a grid of cells, as shown in Figure 11.28. (The rows and columns need not have equal size.) Each checkbox spans two columns, and the text area spans four rows.

.

Figure 11.28: Dialog box grid used in design

To describe the layout to the grid bag manager, use the following procedure:

1. Create an object of type GridBagLayout. You don't need to tell it how many rows and columns the underlying grid has. Instead, the layout manager will try to guess it from the information you give it later.
2. Set this GridBagLayout object to be the layout manager for the component.
3. For each component, create an object of type GridBagConstraints. Set field values of the

GridBagConstraints object to specify how the components are laid out within the grid bag.

4. Finally, add each component with its constraints by using the call

   add(component, constraints);

Here's an example of the code needed. (We'll go over the various constraints in more detail in the sections that follow—so don't worry if you don't know what some of the constraints do.)

```
var layout = new GridBagLayout();
panel.setLayout(layout);
var constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;
constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
panel.add(component, constraints);
```

The trick is knowing how to set the state of the GridBagConstraints object. We'll discuss this object in the sections that follow.

11.6.2. The gridx, gridy, gridwidth, and gridheight Parameters

The gridx, gridy, gridwidth, and gridheight constraints define where the component is located in the grid. The gridx and gridy values specify the column and row positions of the upper left corner of the component to be added. The gridwidth and gridheight values determine how many columns and rows the component occupies.

The top-left corner has gridx = 0 and gridy = 0. The text area in our example has gridx = 2, gridy = 0 because it starts in column 2 (that is, the third column) of row 0. It has gridwidth = 1 and

gridheight = 4 because it spans one column and four rows.

11.6.3. Weight Fields

You always need to set the weight fields (weightx and weighty) for each area in a grid bag layout. If you set the weight to 0, the area never grows or shrinks beyond its initial size in that direction. In the grid bag layout for Figure 11.27, we set the weightx field of the labels to be 0. This allows the labels to keep constant width when you resize the window. On the other hand, if you set the weights for all areas to 0, the container will huddle in the center of its allotted area instead of stretching to fill it.

Conceptually, the problem with the weight parameters is that weights are properties of rows and columns, not individual cells. But you need to specify them for cells because the grid bag layout does not expose the rows and columns. The row and column weights are computed as the maxima of the cell weights in each row or column. Thus, if you want a row or column to stay at a fixed size, you need to set the weights of all components in it to zero.

Note that the weights don't actually give the relative sizes of the columns. They tell what proportion of the "slack" space should be allocated to each area if the container exceeds its preferred size. This isn't particularly intuitive. I recommend that you set all weights at 100. Then, run the program and see how the layout looks. Resize the dialog to see how the rows and columns adjust. If you find that a particular row or column should not grow, set the weights of all components in it to zero. You can tinker with other weight values, but it is usually not worth the effort.

11.6.4. The fill and anchor Parameters

If you don't want a component to stretch out and fill the entire area, set the fill constraint. You have four possibilities for this parameter: the valid values are GridBagConstraints.NONE, GridBagConstraints.HORIZONTAL,

GridBagConstraints.VERTICAL, and GridBagConstraints.BOTH.

If the component does not fill the entire area, you can specify where in the area you want it by setting the anchor field. The valid values are GridBagConstraints.CENTER (the default), GridBagConstraints.NORTH, GridBagConstraints.NORTHEAST, GridBagConstraints.EAST, and so on.

11.6.5. Padding

You can surround a component with additional blank space by setting the insets field of GridBagConstraints. Set the left, top, right, and bottom values of the Insets object to the amount of space that you want to have around the component. This is called the external padding.

The ipadx and ipady values set the internal padding. These values are added to the minimum width and height of the component. This ensures that the component does not shrink down to its minimum size.

11.6.6. Alternative Method to Specify the gridx, gridy, gridwidth, and gridheight Parameters

The AWT documentation recommends that instead of setting the gridx and gridy values to absolute positions, you set them to the constant GridBagConstraints.RELATIVE. Then, add the components to the grid bag layout in a standardized order, going from left to right in the first row, then moving along the next row, and so on.

You would still specify the number of rows and columns spanned, by giving the appropriate gridheight and gridwidth fields. However, if the component extends to the last row or column, you don't need to specify the actual number, but the constant GridBagConstraints.REMAINDER. This tells the layout manager that the component is the last one in its row.

This scheme does seem to work. But it sounds really goofy to hide the actual placement information from the layout

manager and hope that it will rediscover it.

11.6.7. A Grid Bag Layout Recipe

In practice, the following recipe makes grid bag layouts relatively trouble-free:

1. Sketch out the component layout on a piece of paper.
2. Find a grid such that the small components are each contained in a cell and the larger components span multiple cells.
3. Label the rows and columns of your grid with 0, 1, 2, 3, . . . You can now read off the gridx, gridy, gridwidth, and gridheight values.
4. For each component, ask yourself whether it needs to fill its cell horizontally or vertically. If not, how do you want it aligned? This tells you the fill and anchor parameters.
5. Set all weights to 100. However, if you want a particular row or column to always stay at its default size, set the weightx or weighty to 0 in all components that belong to that row or column.
6. Write the code. Carefully double-check your settings for the GridBagConstraints. One wrong constraint can ruin your whole layout.
7. Compile, run, and enjoy.

11.6.8. A Helper Class to Tame the Grid Bag Constraints

The most tedious aspect of the grid bag layout is writing the code that sets the constraints. Most programmers write helper functions or a small helper class for this purpose. I present such a class after the complete code for the font dialog example. This class has the following features:

- Its name is short: GBC instead of GridBagConstraints.
- It extends GridBagConstraints, so you can use shorter

names such as GBC.EAST for the constants.

- Use a GBC object when adding a component, such as

```
add(component, new GBC(1, 2));
```

- There are two constructors to set the most common parameters: gridx and gridy, or gridx, gridy, gridwidth, and gridheight.

```
add(component, new GBC(1, 2, 1, 4));
```

- There are convenient setters for the fields that come in x/y pairs:

```
add(component, new GBC(1, 2).setWeight(100, 100));
```

- The setter methods return this, so you can chain them:

```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setWeigh
```

- The setInsets methods construct the Insets object for you. To get one-pixel insets, simply call

```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setInsets(
```

Listing 11.7 shows the frame class for the font dialog example. The GBC helper class is in Listing 11.8. Here is the code that adds the components to the grid bag:

```
add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0).
add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0).s
add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100
add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100
add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100
```

Once you understand the grid bag constraints, this kind of code is fairly easy to read and debug.

Listing 11.7 **v2ch11/gridbag/FontFrame.java**

```
package v2ch11.gridbag;

import module java.desktop;

/**
 * A frame that uses a grid bag layout to arrange font selection components.
 */
public class FontFrame extends JFrame {
   private static final int TEXT_ROWS = 10;
   private static final int TEXT_COLUMNS = 20;

   private JComboBox<String> face;
   private JComboBox<Integer> size;
   private JCheckBox bold;
   private JCheckBox italic;
   private JTextArea sample;

   public FontFrame() {
      var layout = new GridBagLayout();
      setLayout(layout);

      ActionListener listener = _ -> updateSample();

      // construct components

      var faceLabel = new JLabel("Face: ");

      face = new JComboBox<>(
            new String[] { "Serif", "SansSerif", "Monospaced", "Dialog", "DialogInp

      face.addActionListener(listener);

      var sizeLabel = new JLabel("Size: ");

      size = new JComboBox<>(new Integer[] { 8, 10, 12, 15, 18, 24, 36, 48 });

      size.addActionListener(listener);

      bold = new JCheckBox("Bold");
      bold.addActionListener(listener);

      italic = new JCheckBox("Italic");
      italic.addActionListener(listener);
```

```
      sample = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
      sample.setText("The quick brown fox jumps over the lazy dog");
      sample.setEditable(false);
      sample.setLineWrap(true);
      sample.setBorder(BorderFactory.createEtchedBorder());

      // add components to grid, using GBC convenience class

      add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
      add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0).setIns
      add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
      add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0).setIns
      add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100)
      add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100
      add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
      pack();
      updateSample();
   }

   public void updateSample() {
      var fontFace = (String) face.getSelectedItem();
      int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
            + (italic.isSelected() ? Font.ITALIC : 0);
      int fontSize = size.getItemAt(size.getSelectedIndex());
      var font = new Font(fontFace, fontStyle, fontSize);
      sample.setFont(font);
      sample.repaint();
   }
}
```

Listing 11.8 **v2ch11/gridbag/GBC.java**

```
package v2ch11.gridbag;

import module java.desktop;

/**
 * This class simplifies the use of the GridBagConstraints class.
 */
public class GBC extends GridBagConstraints {
   /**
    * Constructs a GBC with a given gridx and gridy position and all other grid
    * values set to the default.
```

```
 * @param gridx the gridx position
 * @param gridy the gridy position
 */
public GBC(int gridx, int gridy) {
   this.gridx = gridx;
   this.gridy = gridy;
}

/**
 * Constructs a GBC with given gridx, gridy, gridwidth, gridheight and all oth
 * constraint values set to the default.
 * @param gridx the gridx position
 * @param gridy the gridy position
 * @param gridwidth the cell span in x-direction
 * @param gridheight the cell span in y-direction
 */
public GBC(int gridx, int gridy, int gridwidth, int gridheight) {
   this.gridx = gridx;
   this.gridy = gridy;
   this.gridwidth = gridwidth;
   this.gridheight = gridheight;
}

/**
 * Sets the anchor.
 * @param anchor the anchor value
 * @return this object for further modification
 */
public GBC setAnchor(int anchor) {
   this.anchor = anchor;
   return this;
}

/**
 * Sets the fill direction.
 * @param fill the fill direction
 * @return this object for further modification
 */
public GBC setFill(int fill) {
   this.fill = fill;
   return this;
}

/**
```

```
     * Sets the cell weights.
     * @param weightx the cell weight in x-direction
     * @param weighty the cell weight in y-direction
     * @return this object for further modification
     */
    public GBC setWeight(double weightx, double weighty) {
       this.weightx = weightx;
       this.weighty = weighty;
       return this;
    }

    /**
     * Sets the insets of this cell.
     * @param distance the spacing to use in all directions
     * @return this object for further modification
     */
    public GBC setInsets(int distance) {
       this.insets = new Insets(distance, distance, distance, distance);
       return this;
    }

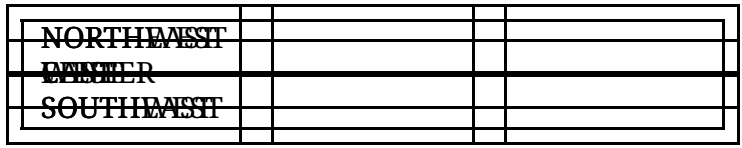
    /**
     * Sets the insets of this cell.
     * @param top the spacing to use on top
     * @param left the spacing to use to the left
     * @param bottom the spacing to use on the bottom
     * @param right the spacing to use to the right
     * @return this object for further modification
     */
    public GBC setInsets(int top, int left, int bottom, int right) {
       this.insets = new Insets(top, left, bottom, right);
       return this;
    }

    /**
     * Sets the internal padding.
     * @param ipadx the internal padding in x-direction
     * @param ipady the internal padding in y-direction
     * @return this object for further modification
     */
    public GBC setIpad(int ipadx, int ipady) {
       this.ipadx = ipadx;
       this.ipady = ipady;
       return this;
```

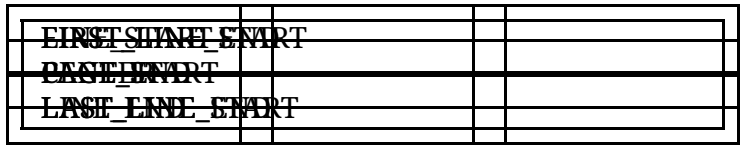
```
   }
}
```

java.awt.GridBagConstraints 1.0

- int gridx
  int gridy
  specify the starting column and row of the cell. The default is 0.
- int gridwidth
  int gridheight
  specify the column and row extent of the cell. The default is 1.
- double weightx
  double weighty
  specify the capacity of the cell to grow. The default is 0.
- int anchor
  indicates the alignment of the component inside the cell. You can choose between absolute positions:



  or their orientation-independent counterparts:



  Use the latter if your application may be localized for right-to-left or top-to-bottom text. The default is CENTER.
- int fill
  specifies the fill behavior of the component inside the cell: one of NONE, BOTH, HORIZONTAL, or VERTICAL. The default is NONE.
- int ipadx
  int ipady
  specify the “internal” padding around the component. The default is 0.
- Insets insets

specifies the “external” padding along the cell boundaries. The default is no padding.

- GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady)1.2
  constructs a GridBagConstraints with all its fields specified in the parameters. This constructor should only be used by automatic code generators because it makes your source code very hard to read.

11.7. Custom Layout Managers

You can design your own LayoutManager class that manages components in a special way. As a fun example, let's arrange all components in a container to form a circle (see Figure 11.29).

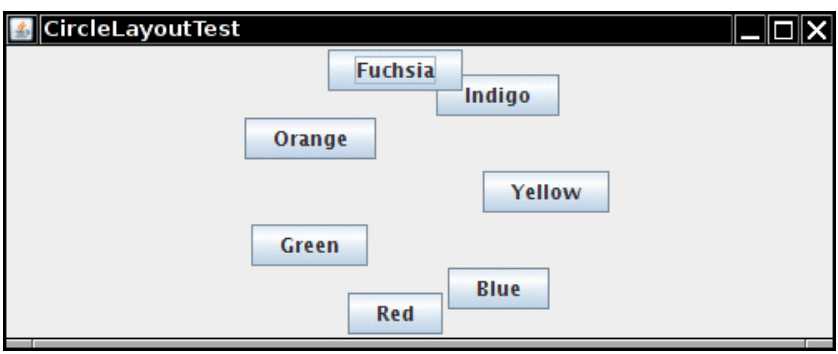


Figure 11.29: Circle layout

Your own layout manager must implement the LayoutManager interface. You need to override the following five methods:

```
void addLayoutComponent(String s, Component c)
void removeLayoutComponent(Component c)
Dimension preferredLayoutSize(Container parent)
Dimension minimumLayoutSize(Container parent)
void layoutContainer(Container parent)
```

The first two methods are called when a component is added or removed. If you don't keep any additional information about the components, you can make them do nothing. The next two methods compute the space required for the minimum and the preferred layout of the components. These are usually the same quantity. The fifth method does the actual work and invokes setBounds on all components.

> The AWT has a second interface, called LayoutManager2, with ten methods to implement rather than five. The main point of the LayoutManager2 interface is to allow you to use the add method with constraints. For example, the BorderLayout and GridBagLayout implement the LayoutManager2 interface.

Listing 11.9 shows the code for the CircleLayout manager which, uselessly enough, lays out the components along a circle inside the parent. The frame class of the sample program is in Listing 11.10.

Listing 11.9 **v2ch11/circleLayout/CircleLayout.java**

```
package v2ch11.circleLayout;

import module java.desktop;

/**
 * A layout manager that lays out components along a circle.
 */
public class CircleLayout implements LayoutManager {
   private int minWidth = 0;
   private int minHeight = 0;
   private int preferredWidth = 0;
   private int preferredHeight = 0;
   private boolean sizesSet = false;
   private int maxComponentWidth = 0;
   private int maxComponentHeight = 0;

   public void addLayoutComponent(String name, Component comp) {
```

```
}

public void removeLayoutComponent(Component comp) {
}

public void setSizes(Container parent) {
   if (sizesSet) return;
   int n = parent.getComponentCount();

   preferredWidth = 0;
   preferredHeight = 0;
   minWidth = 0;
   minHeight = 0;
   maxComponentWidth = 0;
   maxComponentHeight = 0;

   // compute the maximum component widths and heights
   // and set the preferred size to the sum of the component sizes
   for (int i = 0; i < n; i++) {
      Component c = parent.getComponent(i);
      if (c.isVisible()) {
         Dimension d = c.getPreferredSize();
         maxComponentWidth = Math.max(maxComponentWidth, d.width);
         maxComponentHeight = Math.max(maxComponentHeight, d.height);
         preferredWidth += d.width;
         preferredHeight += d.height;
      }
   }
   minWidth = preferredWidth / 2;
   minHeight = preferredHeight / 2;
   sizesSet = true;
}

public Dimension preferredLayoutSize(Container parent) {
   setSizes(parent);
   Insets insets = parent.getInsets();
   int width = preferredWidth + insets.left + insets.right;
   int height = preferredHeight + insets.top + insets.bottom;
   return new Dimension(width, height);
}

public Dimension minimumLayoutSize(Container parent) {
   setSizes(parent);
   Insets insets = parent.getInsets();
```

```
      int width = minWidth + insets.left + insets.right;
      int height = minHeight + insets.top + insets.bottom;
      return new Dimension(width, height);
   }

   public void layoutContainer(Container parent) {
      setSizes(parent);

      // compute center of the circle

      Insets insets = parent.getInsets();
      int containerWidth = parent.getSize().width - insets.left - insets.right;
      int containerHeight = parent.getSize().height - insets.top - insets.bottom;

      int xcenter = insets.left + containerWidth / 2;
      int ycenter = insets.top + containerHeight / 2;

      // compute radius of the circle

      int xradius = (containerWidth - maxComponentWidth) / 2;
      int yradius = (containerHeight - maxComponentHeight) / 2;
      int radius = Math.min(xradius, yradius);

      // lay out components along the circle

      int n = parent.getComponentCount();
      for (int i = 0; i < n; i++) {
         Component c = parent.getComponent(i);
         if (c.isVisible()) {
            double angle = 2 * Math.PI * i / n;

            // center point of component
            int x = xcenter + (int) (Math.cos(angle) * radius);
            int y = ycenter + (int) (Math.sin(angle) * radius);

            // move component so that its center is (x, y)
            // and its size is its preferred size
            Dimension d = c.getPreferredSize();
            c.setBounds(x - d.width / 2, y - d.height / 2, d.width, d.height);
         }
      }
   }
}
```

Listing 11.10 **v2ch11/circleLayout/CircleLayoutFrame.java**

```
package v2ch11.circleLayout;

import module java.desktop;

/**
 * A frame that shows buttons arranged along a circle.
 */
public class CircleLayoutFrame extends JFrame {
   public CircleLayoutFrame() {
      setLayout(new CircleLayout());
      add(new JButton("Yellow"));
      add(new JButton("Blue"));
      add(new JButton("Red"));
      add(new JButton("Green"));
      add(new JButton("Orange"));
      add(new JButton("Fuchsia"));
      add(new JButton("Indigo"));
      pack();
   }
}
```

java.awt.LayoutManager1.0

- void addLayoutComponent(String name, Component comp)
  adds a component to the layout.
- void removeLayoutComponent(Component comp)
  removes a component from the layout.
- Dimension preferredLayoutSize(Container cont)
  returns the preferred size dimensions for the container under this layout.
- Dimension minimumLayoutSize(Container cont)
  returns the minimum size dimensions for the container under this layout.
- void layoutContainer(Container cont)
  lays out the components in a container.

11.8. Dialog Boxes

In GUI applications, you usually want separate dialog boxes to pop up to give information to, or get information from, the user.

Just as with most windowing systems, AWT distinguishes between modal and modeless dialog boxes. A modal dialog box won't let users interact with the remaining windows of the application until he or she deals with it. Use a modal dialog box when you need information from the user before you can proceed with execution. For example, when the user wants to read a file, a modal file dialog box is the one to pop up. The user must specify a file name before the program can begin the read operation. Only when the user closes the modal dialog box can the application proceed.

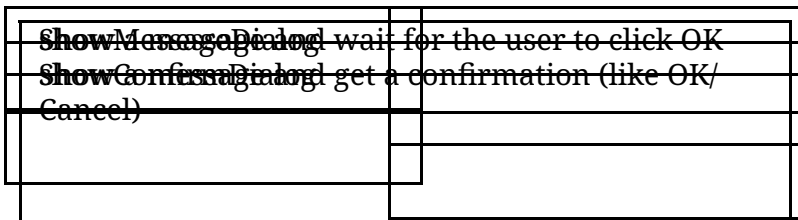
A modeless dialog box lets the user enter information in both the dialog box and the remainder of the application. One example of a modeless dialog is a toolbar. The toolbar can stay in place as long as needed, and the user can interact with both the application window and the toolbar.

Let's start this section with the simplest dialogs—modal dialogs with just a single message. Swing has a convenient JOptionPane class that lets you put up a simple dialog without writing any special dialog box code. Next, you will see how to write more complex dialogs by implementing your own dialog windows. Finally, you will see how to transfer data from your application into a dialog and back.

We'll conclude the discussion of dialog boxes by looking at the Swing JFileChooser.

11.8.1. Option Panes

Swing has a set of ready-made simple dialogs that suffice to ask the user for a single piece of information. The JOptionPane has four static methods to show these simple dialogs:



showOptionDialog Show a message and get a user option (typically from a sequence of buttons)

showInputDialog Show a message and get user input (typically from an input field or combo box)

Figure 11.30 shows a typical dialog. As you can see, the dialog has the following components:

- An icon
- A message
- One or more option buttons



Figure 11.30: A JOptionPane

The input dialog has an additional component for user input. This can be a text field into which the user can type an arbitrary string, or a combo box from which the user can select one item.

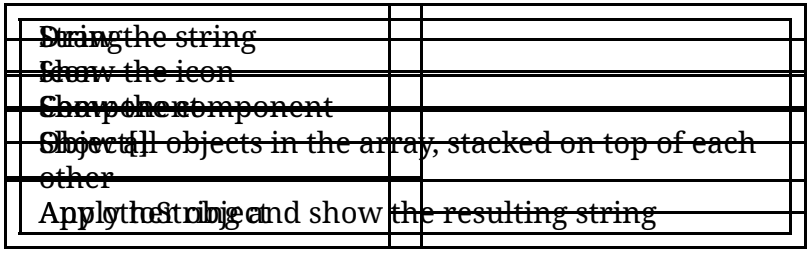
The exact layout of these dialogs and the choice of icons for standard message types depend on the pluggable look-and-feel.

The icon on the left side depends on one of five message types:

ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE

The PLAIN_MESSAGE type has no icon. Each dialog type also has a method that lets you supply your own icon instead.

For each dialog type, you can specify a message. This message can be a string, an icon, a user interface component, or any other object. Here is how the message object is displayed:

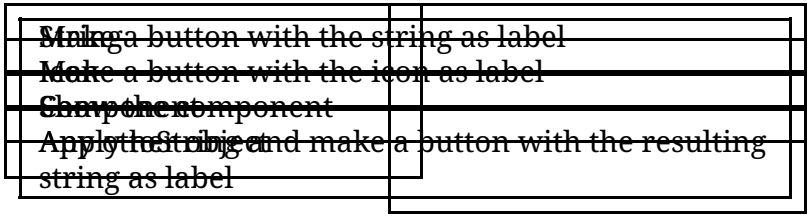


Of course, supplying a message string is by far the most common case. Supplying a Component gives you ultimate flexibility because you can make the paintComponent method draw anything you want.

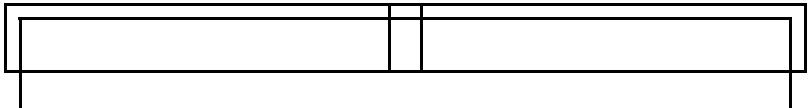
The buttons at the bottom depend on the dialog type and the option type. When calling showMessageDialog and showInputDialog, you get only a standard set of buttons (OK and OK/Cancel, respectively). When calling showConfirmDialog, you can choose among four option types:

DEFAULT_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION

With the showOptionDialog you can specify an arbitrary set of options. You supply an array of objects for the options. Each array element is rendered as follows:


String
Make a button with the string as label
Icon
Make a button with the icon as label
Component
Show the component
Object
Apply toString and make a button with the resulting string as label

The return values of these functions are as follows:



None Information Warning Question Error Custom	No icon An information icon A warning icon A question icon An error icon A custom icon

showMessageDialog	Shows a message
showConfirmDialog	Shows a message and returns an integer representing the chosen option
showOptionDialog	Shows a message and returns an integer representing the chosen option
showInputDialog	Shows a message and returns a string that is the user supplied or selected value

The showConfirmDialog and showOptionDialog return integers to indicate which button the user chose. For the option dialog, this is simply the index of the chosen option or the value CLOSED_OPTION if the user closed the dialog instead of choosing an option. For the confirmation dialog, the return value can be one of the following:

OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION

This all sounds like a bewildering set of choices, but in practice it is simple. Follow these steps:

1. Choose the dialog type (message, confirmation, option, or input).
2. Choose the icon (error, information, warning, question, none, or custom).
3. Choose the message (string, icon, custom component, or a stack of them).
4. For a confirmation dialog, choose the option type (default, Yes/No, Yes/No/Cancel, or OK/Cancel).
5. For an option dialog, choose the options (strings, icons, or custom components) and the default option.
6. For an input dialog, choose between a text field and a combo box.
7. Locate the appropriate method to call in the JOptionPane API.

For example, suppose you want to show the dialog in Figure 11.30. The dialog shows a message and asks the user to confirm

or cancel. Thus, it is a confirmation dialog. The icon is a question icon. The message is a string. The option type is OK_CANCEL_OPTION. Here is the call you would make:

```
int selection = JOptionPane.showConfirmDialog(parent,
   "Message", "Title",
   JOptionPane.OK_CANCEL_OPTION,
   JOptionPane.QUESTION_MESSAGE);
if (selection == JOptionPane.OK_OPTION) . . .
```

The message string can contain newline ('\n') characters. Such a string is displayed in multiple lines.

---

javax.swing.JOptionPane1.2

- static void showMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)
- static void showMessageDialog(Component parent, Object message, String title, int messageType)
- static void showMessageDialog(Component parent, Object message)
- static void showInternalMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)
- static void showInternalMessageDialog(Component parent, Object message, String title, int messageType)
- static void showInternalMessageDialog(Component parent, Object message)
  show a message dialog or an internal message dialog. (An internal dialog is rendered entirely within its owner's frame.) The parent component can be null. The message to show on the dialog can be a string, icon, component, or an array of them. The messageType parameter is one of ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, or PLAIN_MESSAGE.
- static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType,

Icon icon)

- static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)
- static int showConfirmDialog(Component parent, Object message, String title, int optionType)
- static int showConfirmDialog(Component parent, Object message)
- static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)
- static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)
- static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType)
- static int showInternalConfirmDialog(Component parent, Object message)
  show a confirmation dialog or an internal confirmation dialog. (An internal dialog is rendered entirely within its owner's frame.) Returns the option selected by the user (one of OK_OPTION, CANCEL_OPTION, YES_OPTION, NO_OPTION), or CLOSED_OPTION if the user closed the dialog. The parent component can be null. The message to show on the dialog can be a string, icon, component, or an array of them. The messageType parameter is one of ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, or PLAIN_MESSAGE, and optionType is one of DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, or OK_CANCEL_OPTION.
- static int showOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)
- static int showInternalOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)
  show an option dialog or an internal option dialog. (An internal dialog is rendered entirely within its owner's frame.) Returns the index of the option selected by the user, or CLOSED_OPTION if the user canceled the dialog.

The parent component can be null. The message to show on the dialog can be a string, icon, component, or an array of them. The messageType parameter is one of ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, or PLAIN_MESSAGE, and optionType is one of DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, or OK_CANCEL_OPTION. The options parameter is an array of strings, icons, or components.

- static Object showInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)
- static String showInputDialog(Component parent, Object message, String title, int messageType)
- static String showInputDialog(Component parent, Object message)
- static String showInputDialog(Object message)
- static String showInputDialog(Component parent, Object message, Object default)**1.4**
- static String showInputDialog(Object message, Object default)**1.4**
- static Object showInternalInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)
- static String showInternalInputDialog(Component parent, Object message, String title, int messageType)
- static String showInternalInputDialog(Component parent, Object message)
  show an input dialog or an internal input dialog. (An internal dialog is rendered entirely within its owner's frame.) Returns the input string typed by the user, or null if the user canceled the dialog. The parent component can be null. The message to show on the dialog can be a string, icon, component, or an array of them. The messageType parameter is one of ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, or PLAIN_MESSAGE.

11.8.2. Creating Dialogs

In the last section, you saw how to use the JOptionPane class to show a simple dialog. In this section, you will see how to create such a dialog by hand.

Figure 11.31 shows a typical modal dialog box—a program information box that is displayed when the user clicks the About button.

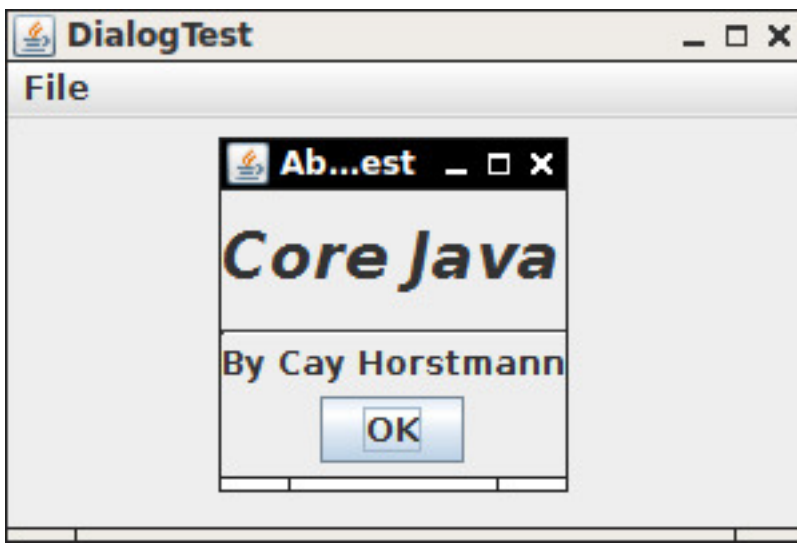


Figure 11.31: An About dialog box

To implement a dialog box, you extend the JDialog class. This is essentially the same process as extending JFrame for the main window for an application. More precisely:

1. In the constructor of your dialog box, call the constructor of the superclass JDialog.
2. Add the user interface components of the dialog box.
3. Add the event handlers.
4. Set the size for the dialog box.

When you call the superclass constructor, you will need to supply the owner frame, the title of the dialog, and the modality.

The owner frame controls where the dialog is displayed. You can supply null as the owner; then, the dialog is owned by a hidden frame.

The modality specifies which other windows of your application are blocked while the dialog is displayed. A modeless dialog does not block other windows. A modal dialog blocks all other windows of the application (except for the children of the dialog). You would use a modeless dialog for a toolbox that the user can always access. On the other hand, you would use a modal dialog if you want to force the user to supply required information before continuing.

Here's the code for a dialog box:

```
public AboutDialog extends JDialog {
   public AboutDialog(JFrame owner) {
      super(owner, "About DialogTest", true);
      add(new JLabel(
         "<html><h1><i>Core Java</i></h1><hr>By Cay Horstmann</ht
         BorderLayout.CENTER);
      var panel = new JPanel();
      var ok = new JButton("OK");

      ok.addActionListener(_ -> setVisible(false));
      panel.add(ok);
      add(panel, BorderLayout.SOUTH);
      setSize(250, 150);
   }
}
```

As you can see, the constructor adds user interface elements—in this case, labels and a button. It adds a handler to the button and sets the size of the dialog.

To display the dialog box, create a new dialog object and make it visible:

```
var dialog = new AboutDialog(this);
dialog.setVisible(true);
```

Actually, in the sample code below, we create the dialog box only once, and we can reuse it whenever the user clicks the About button.

```
if (dialog == null) // first time
   dialog = new AboutDialog(this);
dialog.setVisible(true);
```

When the user clicks the OK button, the dialog box should close. This is handled in the event handler of the OK button:

```
ok.addActionListener(_ -> setVisible(false));
```

When the user closes the dialog by clicking the Close button, the dialog is also hidden. Just as with a JFrame, you can override this behavior with the setDefaultCloseOperation method.

Listing 11.11 is the code for the frame class of the test program. Listing 11.12 shows the dialog class.

Listing 11.11 **v2ch11/dialog/DialogFrame.java**

```
package v2ch11.dialog;

import module java.desktop;

/**
 * A frame with a menu whose File->About action shows a dialog.
 */
public class DialogFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;
   private AboutDialog dialog;

   public DialogFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
      // construct a File menu

      var menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      var fileMenu = new JMenu("File");
      menuBar.add(fileMenu);

      // add About and Exit menu items

      // the About item shows the About dialog

      var aboutItem = new JMenuItem("About");
      aboutItem.addActionListener(_ -> {
         if (dialog == null) // first time
            dialog = new AboutDialog(DialogFrame.this);
         dialog.setVisible(true); // pop up dialog
      });
      fileMenu.add(aboutItem);

      // the Exit item exits the program

      var exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(_ -> System.exit(0));
      fileMenu.add(exitItem);
   }
}
```

Listing 11.12 **v2ch11/dialog/AboutDialog.java**

```
package v2ch11.dialog;

import module java.desktop;

/**
 * A sample modal dialog that displays a message and waits for the user to clic
 */
public class AboutDialog extends JDialog {
   public AboutDialog(JFrame owner) {
      super(owner, "About DialogTest", true);

      // add HTML label to center

      add(new JLabel("<html><h1><i>Core Java</i></h1><hr>By Cay Horstmann
```

```
            BorderLayout.CENTER);

      // OK button closes the dialog

      var ok = new JButton("OK");
      ok.addActionListener(_ -> setVisible(false));

      // add OK button to southern border

      var panel = new JPanel();
      panel.add(ok);
      add(panel, BorderLayout.SOUTH);

      pack();
   }
}
```

javax.swing.JDialog1.2

- public JDialog(Frame parent, String title, boolean modal) constructs a dialog. The dialog is not visible until it is explicitly shown.

11.8.3. Data Exchange

The most common reason to put up a dialog box is to get information from the user. You have already seen how easy it is to make a dialog box object: Give it initial data and call setVisible(true) to display the dialog box on the screen. Now let's see how to transfer data in and out of a dialog box.

Consider the dialog box in Figure 11.32 that could be used to obtain a username and a password to connect to some online service.

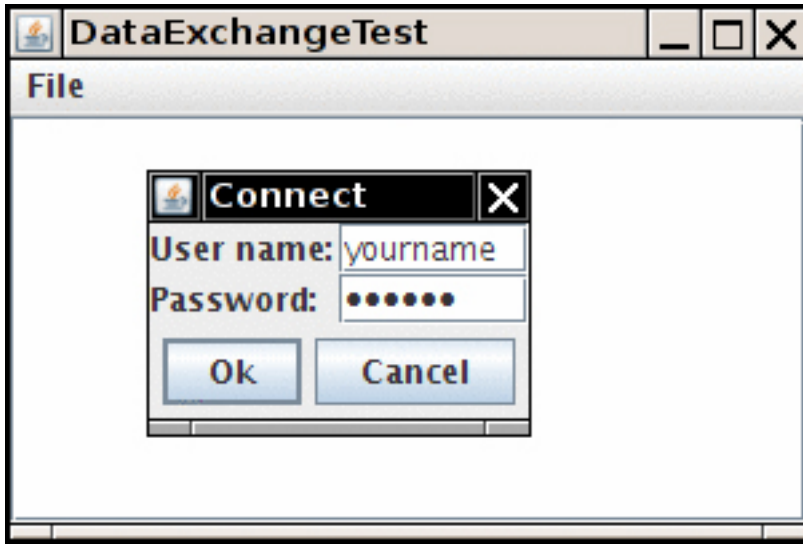


Figure 11.32: Password dialog box

Your dialog box should provide methods to set default data. For example, the PasswordChooser class of the example program has a method, setUser, to place default values into the next fields:

```
public void setUser(User u) {
   username.setText(u.getName());
}
```

Once you set the defaults (if desired), show the dialog by calling setVisible(true). The dialog is now displayed.

The user then fills in the information and clicks the OK or Cancel button. The event handlers for both buttons call setVisible(false), which terminates the call to setVisible(true). Alternatively, the user may close the dialog. If you did not install a window listener for the dialog, the default window closing operation applies: The dialog becomes invisible, which also terminates the call to setVisible(true).

The important issue is that the call to setVisible(true) blocks until the user has dismissed the dialog. This makes it easy to

implement modal dialogs.

You want to know whether the user has accepted or canceled the dialog. Our sample code sets the ok flag to false before showing the dialog. Only the event handler for the OK button sets the ok flag to true; that's how you retrieve the user input from the dialog.

> Transferring data out of a modeless dialog is not as simple. When a modeless dialog is displayed, the call to setVisible(true) does not block and the program continues running while the dialog is displayed. If the user selects items on a modeless dialog and then clicks OK, the dialog needs to send an event to some listener in the program.

The example program contains another useful improvement. When you construct a JDialog object, you need to specify the owner frame. However, quite often you want to show the same dialog with different owner frames. It is better to pick the owner frame when you are ready to show the dialog, not when you construct the PasswordChooser object.

The trick is to have the PasswordChooser extend JPanel instead of JDialog. Build a JDialog object on the fly in the showDialog method:

```
public boolean showDialog(Frame owner, String title) {
   ok = false;

   if (dialog == null || dialog.getOwner() != owner) {
      dialog = new JDialog(owner, true);
      dialog.add(this);
      dialog.pack();
   }

   dialog.setTitle(title);
   dialog.setVisible(true);
   return ok;
}
```

Note that it is safe to have owner equal to null.

You can do even better. Sometimes, the owner frame isn't readily available. It is easy enough to compute it from any parent component, like this:

```
Frame owner;
if (parent instanceof Frame)
   owner = (Frame) parent;
else
   owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, p
```

We use this enhancement in our sample program. The JOptionPane class also uses this mechanism.

Many dialogs have a default button, which is automatically selected if the user presses a trigger key (Enter in most look-and-feel implementations). The default button is specially marked, often with a thick outline.

Set the default button in the root pane of the dialog:

```
dialog.getRootPane().setDefaultButton(okButton);
```

If you follow the suggestion of laying out the dialog in a panel, then you must be careful to set the default button only after you wrapped the panel into a dialog. The panel dialog itself has no root pane.

Listing 11.13 is for the frame class of the program that illustrates the data flow into and out of a dialog box. Listing 11.14 shows the dialog class.

Listing 11.13 **v2ch11/dataExchange/DataExchangeFrame.java**

```
package v2ch11.dataExchange;

import module java.desktop;
```

```
/**
 * A frame with a menu whose File->Connect action shows a password dialog.
 */
public class DataExchangeFrame extends JFrame {
   public static final int TEXT_ROWS = 20;
   public static final int TEXT_COLUMNS = 40;
   private PasswordChooser dialog = null;
   private JTextArea textArea;

   public DataExchangeFrame() {
      // construct a File menu

      var mbar = new JMenuBar();
      setJMenuBar(mbar);
      var fileMenu = new JMenu("File");
      mbar.add(fileMenu);

      // add Connect and Exit menu items

      var connectItem = new JMenuItem("Connect");
      connectItem.addActionListener(new ConnectAction());
      fileMenu.add(connectItem);

      // the Exit item exits the program

      var exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(_ -> System.exit(0));
      fileMenu.add(exitItem);

      textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
      add(new JScrollPane(textArea), BorderLayout.CENTER);
      pack();
   }

   /**
    * The Connect action pops up the password dialog.
    */
   private class ConnectAction implements ActionListener {
      public void actionPerformed(ActionEvent event) {
         // if first time, construct dialog

         if (dialog == null) dialog = new PasswordChooser();

         // set default values
```

```
         dialog.setUser(new User("yourname", null));

         // pop up dialog
         if (dialog.showDialog(DataExchangeFrame.this, "Connect")) {
            // if accepted, retrieve user input
            User u = dialog.getUser();
            textArea.append("user name = " + u.getName() + ", password = "
                  + (new String(u.getPassword())) + "\n");
         }
      }
   }
}
```

Listing 11.14 **v2ch11/dataExchange/PasswordChooser.java**

```
package v2ch11.dataExchange;

import module java.desktop;

/**
 * A password chooser that is shown inside a dialog.
 */
public class PasswordChooser extends JPanel {
   private JTextField username;
   private JPasswordField password;
   private JButton okButton;
   private boolean ok;
   private JDialog dialog;

   public PasswordChooser() {
      setLayout(new BorderLayout());

      // construct a panel with username and password fields

      var panel = new JPanel();
      panel.setLayout(new GridLayout(2, 2));
      panel.add(new JLabel("User name:"));
      panel.add(username = new JTextField(""));
      panel.add(new JLabel("Password:"));
      panel.add(password = new JPasswordField(""));
      add(panel, BorderLayout.CENTER);

      // create Ok and Cancel buttons that terminate the dialog
```

```
    okButton = new JButton("Ok");
    okButton.addActionListener(_ -> {
       ok = true;
       dialog.setVisible(false);
    });

    var cancelButton = new JButton("Cancel");
    cancelButton.addActionListener(_ -> dialog.setVisible(false));

    // add buttons to southern border

    var buttonPanel = new JPanel();
    buttonPanel.add(okButton);
    buttonPanel.add(cancelButton);
    add(buttonPanel, BorderLayout.SOUTH);
}

/**
 * Sets the dialog defaults.
 * @param u the default user information
 */
public void setUser(User u) {
    username.setText(u.getName());
}

/**
 * Gets the dialog entries.
 * @return a User object whose state represents the dialog entries
 */
public User getUser() {
    return new User(username.getText(), password.getPassword());
}

/**
 * Show the chooser panel in a dialog.
 * @param parent a component in the owner frame or null
 * @param title the dialog window title
 */
public boolean showDialog(Component parent, String title) {
    ok = false;

    // locate the owner frame
```

```
      Frame owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, pa

      // if first time, or if owner has changed, make new dialog

      if (dialog == null || dialog.getOwner() != owner) {
         dialog = new JDialog(owner, true);
         dialog.add(this);
         dialog.getRootPane().setDefaultButton(okButton);
         dialog.pack();
      }

      // set title and show dialog

      dialog.setTitle(title);
      dialog.setVisible(true);
      return ok;
   }
}
```

javax.swing.SwingUtilities1.2

- Container getAncestorOfClass(Class c, Component comp)
  returns the innermost parent container of the given component that belongs to the given class or one of its subclasses.

javax.swing.JComponent1.2

- JRootPane getRootPane()
  gets the root pane enclosing this component, or null if this component does not have an ancestor with a root pane.

javax.swing.JRootPane1.2

- void setDefaultButton(JButton button)
  sets the default button for this root pane. To deactivate the default button, call this method with a null argument.

javax.swing.JButton1.2

- boolean isDefaultButton()
  returns true if this button is the default button of its root pane.

11.8.4. File Dialogs

In an application, you often want to be able to open and save files. A good file dialog box that shows files and directories and lets the user navigate the file system is hard to write, and you definitely don't want to reinvent that wheel. Fortunately, Swing provides a JFileChooser class that allows you to display a file dialog box similar to the one that most native applications use. JFileChooser dialogs are always modal. Note that the JFileChooser class is not a subclass of JDialog. Instead of calling setVisible(true), call showOpenDialog to display a dialog for opening a file, or call showSaveDialog to display a dialog for saving a file. The button for accepting a file is then automatically labeled Open or Save. You can also supply your own button label with the showDialog method. Figure 11.33 shows an example of the file chooser dialog box.

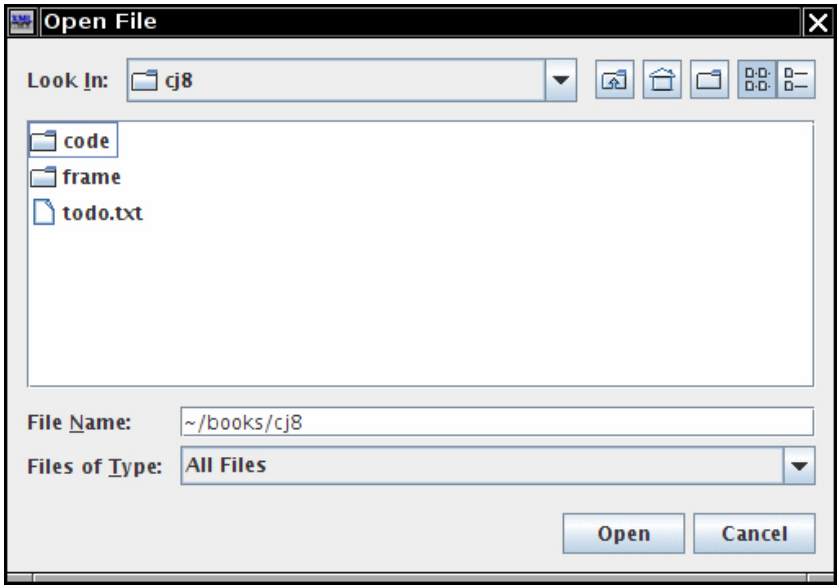


Figure 11.33: A file chooser dialog box

Here are the steps to put up a file dialog box and recover what the user chooses from the box:

1. Make a JFileChooser object. Unlike the constructor for the JDialog class, you do not supply the parent component. This allows you to reuse a file chooser dialog with multiple frames.
   For example:

   ```
   var chooser = new JFileChooser();
   ```

   Reusing a file chooser object is a good idea because the JFileChooser constructor can be quite slow, especially on Windows when the user has many mapped network drives.

   ---

2. Set the directory by calling the setCurrentDirectory method.
   For example, to use the current working directory

```
chooser.setCurrentDirectory(new File("."));
```

you need to supply a File object. File objects are explained in detail in Chapter 2. All you need to know for now is that the constructor File(String filename) turns a file or directory name into a File object.

3. If you have a default file name that you expect the user to choose, supply it with the setSelectedFile method:

```
chooser.setSelectedFile(new File(filename));
```

4. To enable the user to select multiple files in the dialog, call the setMultiSelectionEnabled method. This is, of course, entirely optional and not all that common.

```
chooser.setMultiSelectionEnabled(true);
```

5. If you want to restrict the display of files in the dialog to those of a particular type (for example, all files with extension .gif), you need to set a file filter. We discuss file filters later in this section.
6. By default, a user can select only files with a file chooser. If you want the user to select directories, use the setFileSelectionMode method. Call it with JFileChooser.FILES_ONLY (the default), JFileChooser.DIRECTORIES_ONLY, or JFileChooser.FILES_AND_DIRECTORIES.
7. Show the dialog box by calling the showOpenDialog or showSaveDialog methods. You must supply the parent component in these calls:

```
int result = chooser.showOpenDialog(parent);
```

or

```
int result = chooser.showSaveDialog(parent);
```

The only difference between these calls is the label of the "approve button"—the button that the user clicks to finish

the file selection. You can also call the showDialog method and pass an explicit text for the approve button:

```
int result = chooser.showDialog(parent, "Select");
```

These calls return only when the user has approved, canceled, or dismissed the file dialog. The return value is JFileChooser.APPROVE_OPTION, JFileChooser.CANCEL_OPTION, or JFileChooser.ERROR_OPTION.

8. Get the selected file or files with the getSelectedFile() or getSelectedFiles() method. These methods return either a single File object or an array of File objects. If you just need the name of the file object, call its getPath method. For example:

```
String filename = chooser.getSelectedFile().getPath();
```

For the most part, these steps are simple. The major difficulty with using a file dialog is to specify a subset of files from which the user should choose. For example, suppose the user should choose a GIF image file. Then, the file chooser should only display files with the extension .gif. It should also give the user some kind of feedback that the displayed files are of a particular category, such as “GIF Images.” But the situation can be more complex. If the user should choose a JPEG image file, the extension can be either .jpg or .jpeg. Instead of a way to codify these complexities, the designers of the file chooser provided a more elegant mechanism: to restrict the displayed files, supply an object that extends the abstract class javax.swing.filechooser.FileFilter. The file chooser passes each file to the file filter and displays only those files that the filter accepts.

At the time of this writing, two such subclasses are supplied: the default filter that accepts all files, and a filter that accepts all files with a given extension. However, it is easy to write ad-hoc file filters. Simply implement the two abstract methods of the FileFilter superclass:

```
public boolean accept(File f);
public String getDescription();
```

The first method tests whether a file should be accepted. The second method returns a description of the file type that can be displayed in the file chooser dialog.

An unrelated FileFilter interface in the java.io package has a single method, boolean accept(File f). It is used in the listFiles method of the File class to list files in a directory. I do not know why the designers of Swing didn't extend this interface—perhaps the Java class library had, way back in 1998, become so complex that they were no longer aware of all the standard classes and interfaces.

You will need to resolve the name conflict between these two identically named types if you import both the java.io and the javax.swing.filechooser packages. The simplest remedy is to import javax.swing.filechooser.FileFilter, not javax.swing.filechooser.*.

Once you have a file filter object, use the setFileFilter method of the JFileChooser class to install it into the file chooser object:

```
chooser.setFileFilter(new FileNameExtensionFilter("Image files", "gif
```

You can install multiple filters to the file chooser by calling

```
chooser.addChoosableFileFilter(filter1);
chooser.addChoosableFileFilter(filter2);
. . .
```

The user selects a filter from the combo box at the bottom of the file dialog. By default, the "All files" filter is always present in the combo box. This is a good idea—just in case a user of your program needs to select a file with a nonstandard extension. However, if you want to suppress the "All files" filter, call

```
chooser.setAcceptAllFileFilterUsed(false);
```

If you reuse a single file chooser for loading and saving different file types, call

```
chooser.resetChoosableFilters();
```

to clear any old file filters before adding new ones.

Finally, you can customize the file chooser by providing special icons and file descriptions for each file that the file chooser displays. Do this by supplying an object of a class extending the FileView class in the javax.swing.filechooser package. This is definitely an advanced technique. Normally, you don't need to supply a file view—the pluggable look-and-feel supplies one for you. But if you want to show different icons for special file types, you can install your own file view. You need to extend the FileView class and implement five methods:

```
Icon getIcon(File f)
String getName(File f)
String getDescription(File f)
String getTypeDescription(File f)
Boolean isTraversable(File f)
```

Then, use the setFileView method to install your file view into the file chooser.

The file chooser calls your methods for each file or directory that it wants to display. If your method returns null for the icon, name, or description, the file chooser then consults the default file view of the look-and-feel. That is good, because it means you need to deal only with the file types for which you want to do something different.

The file chooser calls the isTraversable method to decide whether to open a directory when a user clicks on it. Note that this method returns a Boolean object, not a boolean value! This seems weird, but it is actually convenient—if you aren't interested in deviating from the default file view, just return

null. The file chooser will then consult the default file view. In other words, the method returns a Boolean to let you choose among three options: true (Boolean.TRUE), false (Boolean.FALSE), or don't care (null).

The example program contains a simple file view class. That class shows a particular icon whenever a file matches a file filter. We use it to display a palette icon for all image files.

```
class FileIconView extends FileView {
   private FileFilter filter;
   private Icon icon;

   public FileIconView(FileFilter aFilter, Icon anIcon) {
      filter = aFilter;
      icon = anIcon;
   }

   public Icon getIcon(File f) {
      if (!f.isDirectory() && filter.accept(f))
         return icon;
      else return null;
   }
}
```

Install this file view into your file chooser with the setFileView method:

```
chooser.setFileView(new FileIconView(filter,
   new ImageIcon("palette.gif")));
```

The file chooser will then show the palette icon next to all files that pass the filter and use the default file view to show all other files. Naturally, we use the same filter that we set in the file chooser.

Finally, you can customize a file dialog by adding an accessory component. For example, Figure 11.34 shows a preview accessory next to the file list. This accessory displays a thumbnail view of the currently selected file.

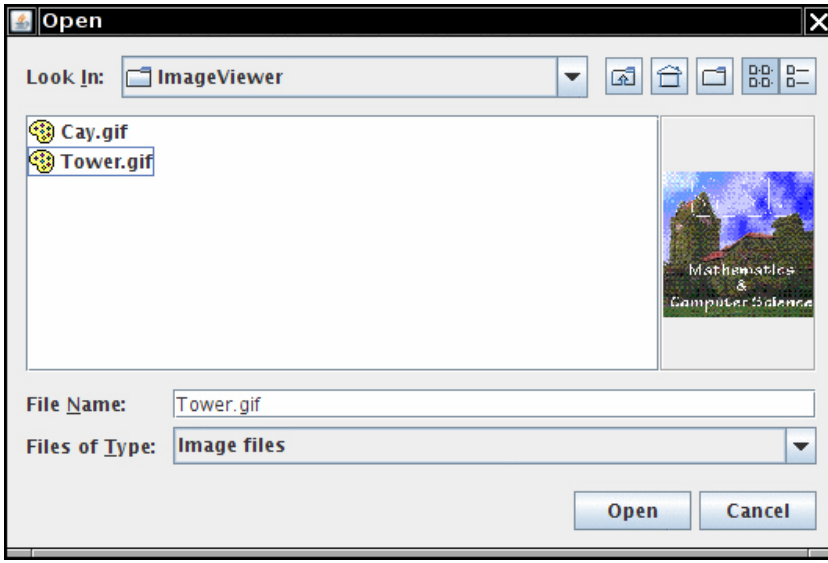


Figure 11.34: A file dialog with a preview accessory

An accessory can be any Swing component. In our case, we extend the JLabel class and set its icon to a scaled copy of the graphics image:

```
class ImagePreviewer extends JLabel {
   public ImagePreviewer(JFileChooser chooser) {
      setPreferredSize(new Dimension(100, 100));
      setBorder(BorderFactory.createEtchedBorder());
   }

   public void loadImage(File f) {
      var icon = new ImageIcon(f.getPath());
      if(icon.getIconWidth() > getWidth())
         icon = new ImageIcon(icon.getImage().getScaledInstance(
            getWidth(), -1, Image.SCALE_DEFAULT));
      setIcon(icon);
      repaint();
   }
}
```

There is just one challenge. We want to update the preview image whenever the user selects a different file. The file

chooser uses the “JavaBeans” mechanism of notifying interested listeners whenever one of its properties changes. The selected file is a property that you can monitor by installing a PropertyChangeListener. Here is the code that you need to trap the notifications:

```
chooser.addPropertyChangeListener(_ -> {
   if (event.getPropertyName() == JFileChooser.SELECTED_FILE_CHA
      var newFile = (File) event.getNewValue();
      // update the accessory
      . . .
   }
});
```

javax.swing.JFileChooser 1.2

- JFileChooser()
  creates a file chooser dialog box that can be used for multiple frames.
- void setCurrentDirectory(File dir)
  sets the initial directory for the file dialog box.
- void setSelectedFile(File file)
- void setSelectedFiles(File[] file)
  set the default file choice for the file dialog box.
- void setMultiSelectionEnabled(boolean b)
  sets or clears the multiple selection mode.
- void setFileSelectionMode(int mode)
  lets the user select files only (the default), directories only, or both files and directories. The mode parameter is one of JFileChooser.FILES_ONLY, JFileChooser.DIRECTORIES_ONLY, and JFileChooser.FILES_AND_DIRECTORIES.
- int showOpenDialog(Component parent)
- int showSaveDialog(Component parent)
- int showDialog(Component parent, String approveButtonText)
  show a dialog in which the approve button is labeled “Open”, “Save”, or with the approveButtonText string. Return APPROVE_OPTION, CANCEL_OPTION (if the user

selected the cancel button or dismissed the dialog), or ERROR_OPTION (if an error occurred).

- File getSelectedFile()
- File[] getSelectedFiles()
  get the file or files that the user selected (or return null if the user didn't select any file).
- void setFileFilter(FileFilter filter)
  sets the file mask for the file dialog box. All files for which filter.accept returns true will be displayed. Also, adds the filter to the list of choosable filters.
- void addChoosableFileFilter(FileFilter filter)
  adds a file filter to the list of choosable filters.
- void setAcceptAllFileFilterUsed(boolean b)
  includes or suppresses an "All files" filter in the filter combo box.
- void resetChoosableFileFilters()
  clears the list of choosable filters. Only the "All files" filter remains unless it is explicitly suppressed.
- void setFileView(FileView view)
  sets a file view to provide information about the files that the file chooser displays.
- void setAccessory(JComponent component)
  sets an accessory component.

javax.swing.filechooser.FileFilter1.2

- boolean accept(File f)
  returns true if the file chooser should display this file.
- String getDescription()
  returns a description of this file filter—for example, "Image files (*.gif,*.jpeg)".

javax.swing.filechooser.FileNameExtensionFilter6

- FileNameExtensionFilter(String description, String... extensions)
  constructs a file filter with the given description that

accepts all directories and all files whose names end in a period followed by one of the given extension strings.

javax.swing.filechooser.FileView1.2

- String getName(File f)
  returns the name of the file f, or null. Normally, this method simply returns f.getName().
- String getDescription(File f)
  returns a human-readable description of the file f, or null. For example, if f is an HTML document, this method might return its title.
- String getTypeDescription(File f)
  returns a human-readable description of the type of the file f, or null. For example, if f is an HTML document, this method might return a string "Hypertext document".
- Icon getIcon(File f)
  returns an icon for the file f, or null. For example, if f is a JPEG file, this method might return a thumbnail icon.
- Boolean isTraversable(File f)
  returns Boolean.TRUE if f is a directory that the user can open. This method might return Boolean.FALSE if a directory is conceptually a compound document. Like all FileView methods, this method can return null to signify that the file chooser should consult the default view instead.

The coverage of GUI and graphics programming continues with the next chapter, which covers more advanced Swing components and sophisticated graphics techniques.

Chapter 12 ▪ Advanced Swing and Graphics

In this chapter, we continue our discussion of the Swing user interface toolkit and AWT graphics from Volume I. We focus on techniques that are applicable to both client-side user interfaces and server-side generation of graphics and images. Swing has sophisticated components for rendering tables and trees. With the 2D graphics API, you can produce vector art of arbitrary complexity. The ImageIO API lets you manipulate raster images. Finally, you can use the printing API to generate printouts and PostScript files.

12.1. Tables

The JTable component displays a two-dimensional grid of objects. Tables are common in user interfaces, and the Swing team has put a lot of effort into the table control. Tables are inherently complex, but—perhaps more successfully than other Swing classes—the JTable component hides much of that complexity. You can produce fully functional tables with rich behavior by writing a few lines of code. You can also write

more code and customize the display and behavior for your specific applications.

In the following sections, I'll explain how to make simple tables, how the user interacts with them, and how to make some of the most common adjustments. As with the other complex Swing controls, it is impossible to cover all aspects in complete detail. For more information, look in Graphic Java™, Third Edition, by David M. Geary (Prentice Hall, 1999), or Core Swing by Kim Topley (Prentice Hall, 1999).

12.1.1. A Simple Table

A JTable does not store its own data but obtains them from a table model. The JTable class has a constructor that wraps a two-dimensional array of objects into a default model. That is the strategy that we use in our first example; later in this chapter, we will turn to table models.

Figure 12.1 shows a typical table, describing the properties of the planets of the solar system. (A planet is gaseous if it consists mostly of hydrogen and helium. You should take the “Color” entries with a grain of salt—that column was added because it will be useful in later code examples.)

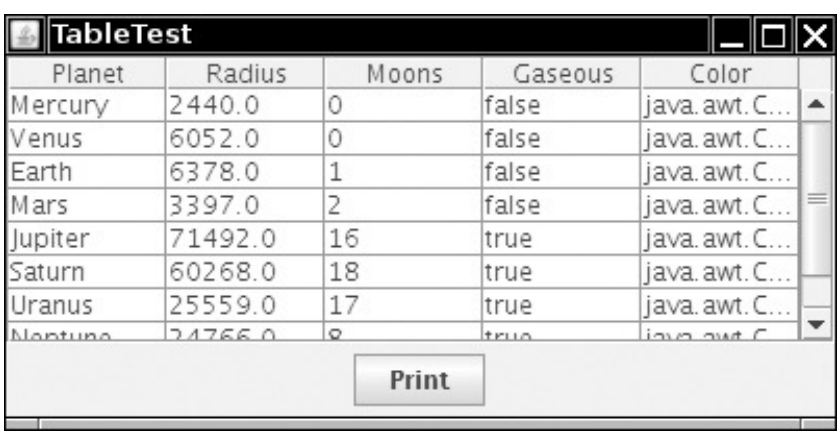


Figure 12.1: A simple table

As you can see from the code in Listing 12.1, the data of the table are stored as a two-dimensional array of Object values:

```
Object[][] cells = {
   { "Mercury", 2440.0, 0, false, Color.YELLOW },
   { "Venus", 6052.0, 0, false, Color.YELLOW },
   . . .
};
```

Here, we take advantage of autoboxing. The entries in the second, third, and fourth columns are automatically converted into objects of type Double, Integer, and Boolean.

The table simply invokes the toString method on each object to display it. That's why the colors show up as java.awt.Color[r=. . .,g=. . .,b=. . .].

Supply the column names in a separate array of strings:

```
String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "C
```

Then, construct a table from the cell and column name arrays:

```
var table = new JTable(cells, columnNames);
```

You can add scroll bars in the usual way—by wrapping the table in a JScrollPane:

```
var pane = new JScrollPane(table);
```

When you scroll the table, the table header doesn't scroll out of view.

Next, click on one of the column headers and drag it to the left or right. See how the entire column becomes detached (see Figure 12.2). You can drop it in a different location. This rearranges the columns in the view only. The data model is not affected.

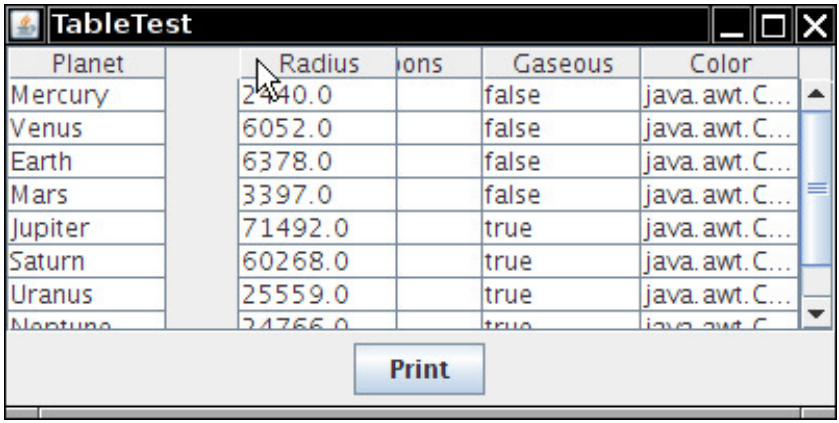


Figure 12.2: Moving a column

To resize columns, simply place the cursor between two columns until the cursor shape changes to an arrow. Then, drag the column boundary to the desired place (see Figure 12.3).

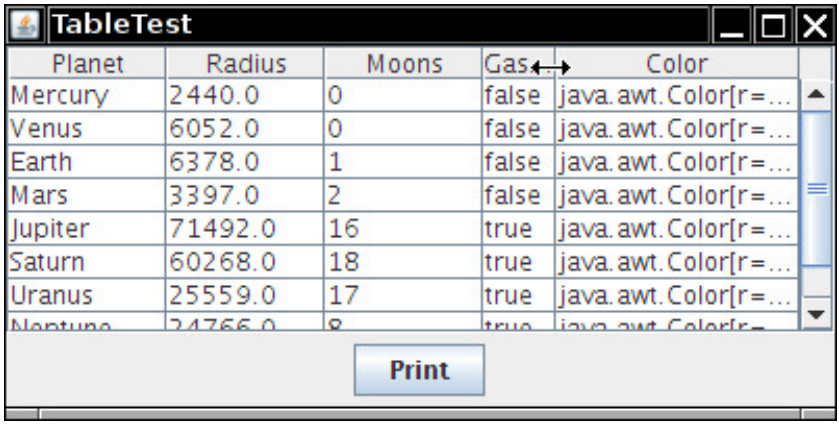


Figure 12.3: Resizing columns

Users can select rows by clicking anywhere in a row. The selected rows are highlighted; you will see later how to get selection events. Users can also edit the table entries by clicking on a cell and typing into it. However, in this code example, the edits do not change the underlying data. In your programs, you should either make cells uneditable or handle cell editing

events and update your model. We will discuss those topics later in this section.

Finally, click on a column header. The rows are automatically sorted. Click again, and the sort order is reversed. This behavior is activated by the call

```
table.setAutoCreateRowSorter(true);
```

You can print a table with the call

```
table.print();
```

If you don't wrap a table into a scroll pane, you need to explicitly add the header:

```
add(table.getTableHeader(), BorderLayout.NORTH);
```

Listing 12.1 **v2ch12/table/TableDemo.java**

```
package v2ch12.table;

import module java.desktop;

/**
 * This program demonstrates how to show a simple table.
 */
class TableDemo {
   void main() {
      EventQueue.invokeLater(() -> {
         var frame = new PlanetTableFrame();
         frame.setTitle("TableTest");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * This frame contains a table of planet data.
```

```
 */
class PlanetTableFrame extends JFrame {
   private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Co
   private Object[][] cells = { { "Mercury", 2440.0, 0, false, Color.YELLOW },
         { "Venus", 6052.0, 0, false, Color.YELLOW },
         { "Earth", 6378.0, 1, false, Color.BLUE }, { "Mars", 3397.0, 2, false, Color.RE
         { "Jupiter", 71492.0, 16, true, Color.ORANGE },
         { "Saturn", 60268.0, 18, true, Color.ORANGE },
         { "Uranus", 25559.0, 17, true, Color.BLUE },
         { "Neptune", 24766.0, 8, true, Color.BLUE },
         { "Pluto", 1137.0, 1, false, Color.BLACK } };

   public PlanetTableFrame() {
      var table = new JTable(cells, columnNames);
      table.setAutoCreateRowSorter(true);
      add(new JScrollPane(table), BorderLayout.CENTER);
      var printButton = new JButton("Print");
      printButton.addActionListener(_ -> {
         try {
            table.print();
         }
         catch (SecurityException | PrinterException e) {
            e.printStackTrace();
         }
      });
      var buttonPanel = new JPanel();
      buttonPanel.add(printButton);
      add(buttonPanel, BorderLayout.SOUTH);
      pack();
   }
}
```

javax.swing.JTable 1.2

- JTable(Object[][] entries, Object[] columnNames)
  constructs a table with a default table model.
- void print() 5.0
  displays a print dialog box and prints the table.
- boolean getAutoCreateRowSorter() 6
- void setAutoCreateRowSorter(boolean newValue) 6
  get or set the autoCreateRowSorter property. The default is false. When set, a default row sorter is automatically set whenever the model changes.

- boolean getFillsViewportHeight()6
- void setFillsViewportHeight(boolean newValue)6
  get or set the fillsViewportHeight property. The default is false. When set, the table always fills the enclosing viewport.

12.1.2. Table Models

In the preceding example, the table data were stored in a two-dimensional array. However, you should generally not use that strategy in your own code. Instead of dumping data into an array to display it as a table, consider implementing your own table model.

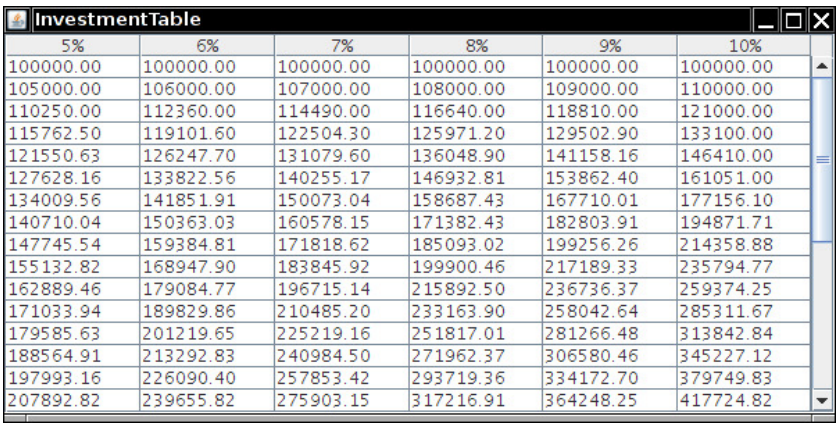
Table models are particularly simple to implement because you can take advantage of the AbstractTableModel class that implements most of the required methods. You only need to supply three methods:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

There are many ways of implementing the getValueAt method. For example, if you want to display the contents of a RowSet that contains the result of a database query, simply provide this method:

```
public Object getValueAt(int r, int c) {
   try {
      rowSet.absolute(r + 1);
      return rowSet.getObject(c + 1);
   }
   catch (SQLException e) {
      e.printStackTrace();
      return null;
   }
}
```

Our sample program is even simpler. We construct a table that shows some computed values—namely, the growth of an investment under different interest rate scenarios (see Figure 12.4).



5%	6%	7%	8%	9%	10%
100000.00	100000.00	100000.00	100000.00	100000.00	100000.00
105000.00	106000.00	107000.00	108000.00	109000.00	110000.00
110250.00	112360.00	114490.00	116640.00	118810.00	121000.00
115762.50	119101.60	122504.30	125971.20	129502.90	133100.00
121550.63	126247.70	131079.60	136048.90	141158.16	146410.00
127628.16	133822.56	140255.17	146932.81	153862.40	161051.00
134009.56	141851.91	150073.04	158687.43	167710.01	177156.10
140710.04	150363.03	160578.15	171382.43	182803.91	194871.71
147745.54	159384.81	171818.62	185093.02	199256.26	214358.88
155132.82	168947.90	183845.92	199900.46	217189.33	235794.77
162889.46	179084.77	196715.14	215892.50	236736.37	259374.25
171033.94	189829.86	210485.20	233163.90	258042.64	285311.67
179585.63	201219.65	225219.16	251817.01	281266.48	313842.84
188564.91	213292.83	240984.50	271962.37	306580.46	345227.12
197993.16	226090.40	257853.42	293719.36	334172.70	379749.83
207892.82	239655.82	275903.15	317216.91	364248.25	417724.82

Figure 12.4: Growth of an investment

The getValueAt method computes the appropriate value and formats it:

```
public Object getValueAt(int r, int c) {
   double rate = (c + minRate) / 100.0;
   int nperiods = r;
   double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, n
   return "%.2f".formatted(futureBalance);
}
```

The getRowCount and getColumnCount methods simply return the number of rows and columns:

```
public int getRowCount() { return years; }
public int getColumnCount() {  return maxRate - minRate + 1; }
```

If you don’t supply column names, the getColumnName method of the AbstractTableModel names the columns A, B, C, and so on. To change the default column names, override the getColumnName method. In this example, we simply label each

column with the interest rate.

```
public String getColumnName(int c) { return (c + minRate) + "%"; }
```

You can find the complete source code in Listing 12.2.

Listing 12.2 **v2ch12/tableModel/InvestmentTable.java**

```
package v2ch12.tableModel;

import module java.desktop;

/**
 * This program shows how to build a table from a table model.
 */
class InvestmentTable {
   void main() {
      EventQueue.invokeLater(() -> {
         var frame = new InvestmentTableFrame();
         frame.setTitle("InvestmentTable");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * This frame contains the investment table.
 */
class InvestmentTableFrame extends JFrame {
   public InvestmentTableFrame() {
      var model = new InvestmentTableModel(30, 5, 10);
      var table = new JTable(model);
      add(new JScrollPane(table));
      pack();
   }
}

/**
 * This table model computes the cell entries each time they are requested. The
 * shows the growth of an investment for a number of years under different in
 */
```

```
class InvestmentTableModel extends AbstractTableModel {
   private static double INITIAL_BALANCE = 100000.0;

   private int years;
   private int minRate;
   private int maxRate;

   /**
    * Constructs an investment table model.
    * @param y the number of years
    * @param r1 the lowest interest rate to tabulate
    * @param r2 the highest interest rate to tabulate
    */
   public InvestmentTableModel(int y, int r1, int r2) {
      years = y;
      minRate = r1;
      maxRate = r2;
   }

   public int getRowCount() {
      return years;
   }

   public int getColumnCount() {
      return maxRate - minRate + 1;
   }

   public Object getValueAt(int r, int c) {
      double rate = (c + minRate) / 100.0;
      int nperiods = r;
      double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods)
      return "%.2f".formatted(futureBalance);
   }

   public String getColumnName(int c) {
      return (c + minRate) + "%";
   }
}
```

javax.swing.table.TableModel 1.2

- int getRowCount()
- int getColumnCount()

  get the number of rows and columns in the table model.

- Object getValueAt(int row, int column)
  gets the value at the given row and column.
- void setValueAt(Object newValue, int row, int column)
  sets a new value at the given row and column.
- boolean isCellEditable(int row, int column)
  returns true if the cell at the given row and column is editable.
- String getColumnName(int column)
  gets the column title.

12.2. Working with Rows and Columns

In the following subsections, you will see how to manipulate the rows and columns in a table. As you read through this material, keep in mind that a Swing table is quite asymmetric—the operations that you can carry out on rows and columns are different. The table component was optimized to display rows of information with the same structure, such as the result of a database query, not an arbitrary two-dimensional grid of objects. You will see this asymmetry throughout this subsection.

12.2.1. Column Classes

In the next example, we again display our planet data, but this time we want to give the table more information about the column types. This is achieved by defining the method

```
Class<?> getColumnClass(int columnIndex)
```

of the table model to return the class that describes the column type.

The JTable class uses this information to pick an appropriate renderer for the class. Table 12.1 shows the default rendering actions.

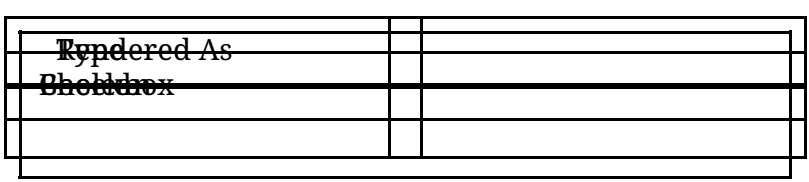


Image		
Object		

Table 12.1: Default Rendering Actions

You can see the checkboxes and images in Figure 12.5. (Thanks to Jim Evins for providing the planet images!)

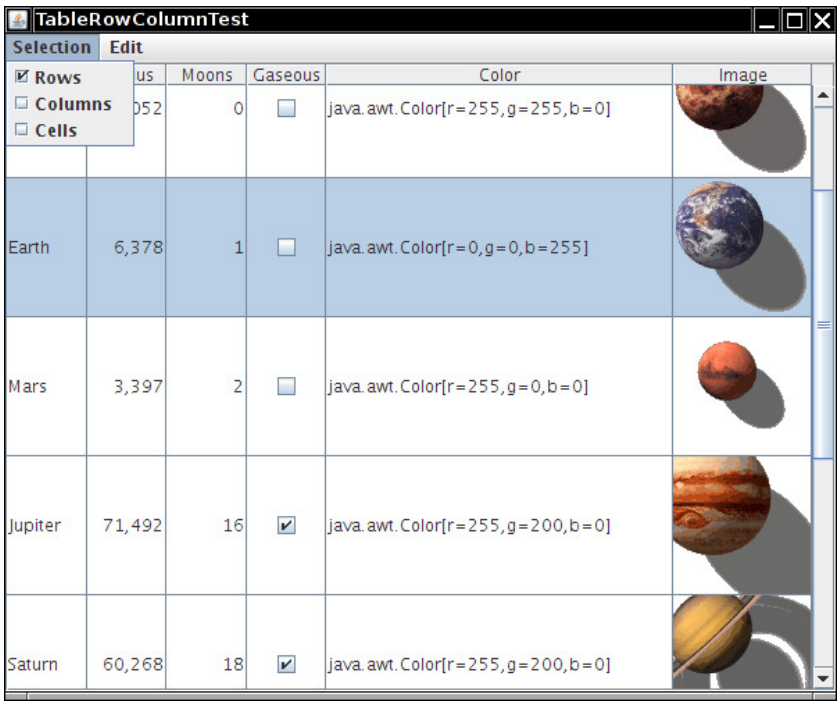


Figure 12.5: A table with planet data

To render other types, you can install a custom renderer—see Section 12.3.

12.2.2. Accessing Table Columns

The JTable class stores information about table columns in objects of type TableColumn. A TableColumnModel object manages the columns. (Figure 12.6 shows the relationships

among the most important table classes.) If you don't want to insert or remove columns dynamically, you won't use the column model much. The most common use for the column model is simply to get a TableColumn object:

```
int columnIndex = . . .;
TableColumn column = table.getColumnModel().getColumn(columnI
```

.

Figure 12.6: Relationships between table classes

12.2.3. Resizing Columns

The TableColumn class gives you control over the resizing behavior of columns. You can set the preferred, minimum, and maximum width with the methods

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

This information is used by the table component to lay out the columns.

Use the method

```
void setResizable(boolean resizable)
```

to control whether the user is allowed to resize the column.

You can programmatically resize a column with the method

```
void setWidth(int width)
```

When a column is resized, the default is to leave the total size of the table unchanged. Of course, the width increase or decrease of the resized column must then be distributed over other columns. The default behavior is to change the size of all columns to the right of the resized column. That's a good default because it allows a user to adjust all columns to a

desired width, moving from left to right.

You can set another behavior from Table 12.2 by using the method

```
void setAutoResizeMode(int mode)
```

of the JTable class.

Mode	Behavior
AUTO_RESIZE_OFF	Don't resize columns; change the table width.
AUTO_RESIZE_NEXT_COLUMN	Resize the next column to compensate; this is the default behavior.
AUTO_RESIZE_SUBSEQUENT_COLUMNS	Resize all subsequent columns equally.
AUTO_RESIZE_LAST_COLUMN	Resize the last column only.
AUTO_RESIZE_ALL_COLUMNS	Resize all columns; this is not a good choice because it prevents the user from adjusting multiple columns to a desired size.

Table 12.2: Resize Modes

12.2.4. Resizing Rows

Row heights are managed directly by the JTable class. If your cells are taller than the default, you may want to set the row height:

```
table.setRowHeight(height);
```

By default, all rows of the table have the same height. You can set the heights of individual rows with the call

```
table.setRowHeight(row, height);
```

The actual row height equals the row height set with these methods, reduced by the row margin. The default row margin is 1 pixel, but you can change it with the call

```
table.setRowMargin(margin);
```

12.2.5. Selecting Rows, Columns, and Cells

Depending on the selection mode, the user can select rows, columns, or individual cells in the table. By default, row selection is enabled. Clicking inside a cell selects the entire row (see Figure 12.5). Call

```
table.setRowSelectionAllowed(false);
```

to disable row selection.

When row selection is enabled, you can control whether the user is allowed to select a single row, a contiguous set of rows, or any set of rows. You need to retrieve the selection model and use its setSelectionMode method:

```
table.getSelectionModel().setSelectionMode(mode);
```

Here, mode is one of the three values:

```
ListSelectionModel.SINGLE_SELECTION
ListSelectionModel.SINGLE_INTERVAL_SELECTION
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

Column selection is disabled by default. You can turn it on with the call

```
table.setColumnSelectionAllowed(true);
```

Enabling both row and column selection is equivalent to enabling cell selection. The user then selects ranges of cells (see Figure 12.7). You can also enable that setting with the call

```
table.setCellSelectionEnabled(true);
```

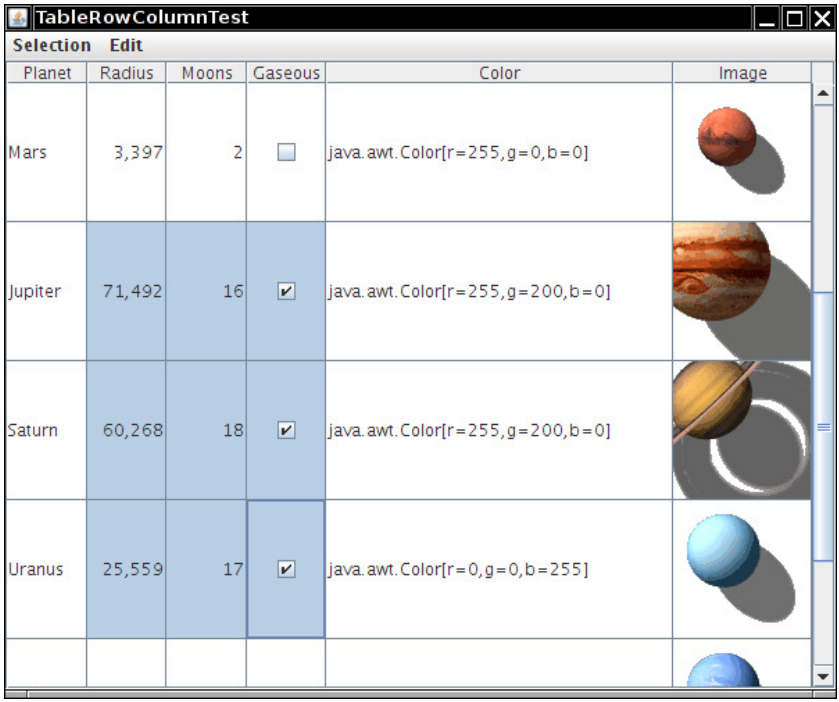


Figure 12.7: Selecting a range of cells

Run the program in Listing 12.3 to watch cell selection in action. Enable row, column, or cell selection in the Selection menu and watch how the selection behavior changes.

You can find out which rows and columns are selected by calling the getSelectedRows and getSelectedColumns methods. Both return an int[] array of the indexes of the selected items. Note that the index values are those of the table view, not the underlying table model. Try selecting rows and columns, then drag columns to different places and sort the rows by clicking on column headers. Use the Print Selection menu item to see which rows and columns are reported as selected.

If you need to translate the table index values to table model index values, use the JTable methods convertRowIndexToModel and convertColumnIndexToModel.

12.2.6. Sorting Rows

As you have seen in our first table example, it is easy to add row sorting to a JTable simply by calling the setAutoCreateRowSorter method. However, to have finer-grained control over the sorting behavior, install a TableRowSorter<M> object into a JTable and customize it. The type parameter M denotes the table model; it needs to be a subtype of the TableModel interface.

```
var sorter = new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);
```

Some columns should not be sortable, such as the image column in our planet data. Turn sorting off by calling

```
sorter.setSortable(IMAGE_COLUMN, false);
```

You can install a custom comparator for each column. In our example, we will sort the colors in the Color column by preferring blue and green over red. When you click on the Color column, you will see that the blue planets go to the bottom of the table. This is achieved with the following call:

```
sorter.setComparator(COLOR_COLUMN, new Comparator<Color>() {
   public int compare(Color c1, Color c2) {
      int d = c1.getBlue() - c2.getBlue();
      if (d != 0) return d;
      d = c1.getGreen() - c2.getGreen();
      if (d != 0) return d;
      return c1.getRed() - c2.getRed();
   }
});
```

If you do not specify a comparator for a column, the sort order is determined as follows:

1. If the column class is String, use the default collator returned by Collator.getInstance(). It sorts strings in a way that is appropriate for the current locale. (See Chapter 7 for more information about locales and collators.)

2. If the column class implements Comparable, use its compareTo method.
3. If a TableStringConverter has been set for the sorter, sort the strings returned by the converter's toString method with the default collator. If you want to use this approach, define a converter as follows:

```
sorter.setStringConverter(new TableStringConverter() {
   public String toString(TableModel model, int row, int column
      Object value = model.getValueAt(row, column);
      convert value to a string and return it
   }
});
```

4. Otherwise, call the toString method on the cell values and sort them with the default collator.

12.2.7. Filtering Rows

In addition to sorting rows, the TableRowSorter can also selectively hide rows—a process called filtering. To activate filtering, set a RowFilter. For example, to include all rows that contain at least one moon, call

```
sorter.setRowFilter(RowFilter.numberFilter(ComparisonType.NOT_E
```

Here, we use a predefined number filter. To construct a number filter, supply

- The comparison type (one of EQUAL, NOT_EQUAL, AFTER, or BEFORE).
- An object of a subclass of Number (such as an Integer or Double). Only objects that have the same class as the given Number object are considered.
- Zero or more column index values. If no index values are supplied, all columns are searched.

The static RowFilter.dateFilter method constructs a date filter in the same way; you need to supply a Date object instead of the Number object.

Finally, the static RowFilter.regexFilter method constructs a filter that looks for strings matching a regular expression. For example,

```
sorter.setRowFilter(RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN
```

only displays those planets whose name doesn't end with an "s". (See Chapter 2 for more information on regular expressions.)

You can also combine filters with the andFilter, orFilter, and notFilter methods. To filter for planets not ending in an "s" with at least one moon, you can use this filter combination:

```
sorter.setRowFilter(RowFilter.andFilter(List.of(
   RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN),
   RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_
```

To implement your own filter, provide a subclass of RowFilter and implement an include method to indicate which rows should be displayed. This is easy to do, but the glorious generality of the RowFilter class makes it a bit scary.

The RowFilter<M, I> class has two type parameters—the types for the model and for the row identifier. When dealing with tables, the model is always a subtype of TableModel and the identifier type is Integer. (At some point in the future, other components might also support row filtering. For example, to filter rows in a JTree, one might use a RowFilter<TreeModel, TreePath>.)

A row filter must implement the method

```
public boolean include(RowFilter.Entry<? extends M, ? extends I> en
```

The RowFilter.Entry class supplies methods to obtain the model, the row identifier, and the value at a given index. Therefore, you can filter both by row identifier and by the contents of the row.

For example, this filter displays every other row:

```
var filter = new RowFilter<TableModel, Integer>() {
   public boolean include(Entry<? extends TableModel, ? extends Int
      return entry.getIdentifier() % 2 == 0;
   }
};
```

If you wanted to include only those planets with an even number of moons, you would instead test for

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

In our sample program, we allow the user to hide arbitrary rows. We store the hidden row indexes in a set. The row filter shows all rows whose indexes are not in that set.

The filtering mechanism wasn't designed for filters with criteria changing over time. In our sample program, we keep calling

```
sorter.setRowFilter(filter);
```

whenever the set of hidden rows changes. Setting a filter causes it to be applied immediately.

12.2.8. Hiding and Displaying Columns

As you saw in the preceding section, you can filter table rows by either their contents or their row identifier. Hiding table columns uses a completely different mechanism.

The removeColumn method of the JTable class removes a column from the table view. The column data are not actually removed from the model—they are just hidden from view. The removeColumn method has a parameter of type TableColumn. If you have the column number (for example, from a call to getSelectedColumns), you need to ask the table model for the actual table column object:

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn(i);
table.removeColumn(column);
```

If you remember the column, you can later add it back in:

```
table.addColumn(column);
```

This method adds the column to the end. If you want it to appear elsewhere, call the moveColumn method.

You can also add a new column that corresponds to a column index in the table model, by adding a new TableColumn object:

```
table.addColumn(new TableColumn(modelColumnIndex));
```

You can have multiple table columns that view the same column of the model.

The program in Listing 12.3 demonstrates selection and filtering of rows and columns.

Listing 12.3 **v2ch12/tableRowColumn/PlanetTableFrame.java**

```
package v2ch12.tableRowColumn;

import module java.base;
import module java.desktop;

/**
 * This frame contains a table of planet data.
 */
public class PlanetTableFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 600;
   private static final int DEFAULT_HEIGHT = 500;

   public static final int COLOR_COLUMN = 4;
   public static final int IMAGE_COLUMN = 5;

   private JTable table;
   private HashSet<Integer> removedRowIndices;
   private ArrayList<TableColumn> removedColumns;
   private JCheckBoxMenuItem rowsItem;
   private JCheckBoxMenuItem columnsItem;
   private JCheckBoxMenuItem cellsItem;
```

```
private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Co
      "Image" };

private Object[][] cells = {
      { "Mercury", 2440.0, 0, false, Color.YELLOW,
            new ImageIcon("images/Mercury.gif") },
      { "Venus", 6052.0, 0, false, Color.YELLOW,
            new ImageIcon("images/Venus.gif") },
      { "Earth", 6378.0, 1, false, Color.BLUE,
            new ImageIcon("images/Earth.gif") },
      { "Mars", 3397.0, 2, false, Color.RED,
            new ImageIcon("images/Mars.gif") },
      { "Jupiter", 71492.0, 16, true, Color.ORANGE,
            new ImageIcon("images/Jupiter.gif") },
      { "Saturn", 60268.0, 18, true, Color.ORANGE,
            new ImageIcon("images/Saturn.gif") },
      { "Uranus", 25559.0, 17, true, Color.BLUE,
            new ImageIcon("images/Uranus.gif") },
      { "Neptune", 24766.0, 8, true, Color.BLUE,
            new ImageIcon("images/Neptune.gif") },
      { "Pluto", 1137.0, 1, false, Color.BLACK,
            new ImageIcon("images/Pluto.gif") } };

public PlanetTableFrame() {
   setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

   var model = new DefaultTableModel(cells, columnNames) {
      public Class<?> getColumnClass(int c) {
         return cells[0][c].getClass();
      }
   };

   table = new JTable(model);

   table.setRowHeight(100);
   table.getColumnModel().getColumn(COLOR_COLUMN).setMinWidth(250);
   table.getColumnModel().getColumn(IMAGE_COLUMN).setMinWidth(100);

   var sorter = new TableRowSorter<TableModel>(model);
   table.setRowSorter(sorter);
   sorter.setComparator(COLOR_COLUMN, Comparator.comparing(Color::get
         .thenComparing(Color::getGreen).thenComparing(Color::getRed));
   sorter.setSortable(IMAGE_COLUMN, false);
   add(new JScrollPane(table), BorderLayout.CENTER);
```

```
removedRowIndices = new HashSet<>();
removedColumns = new ArrayList<>();

var filter = new RowFilter<TableModel, Integer>() {
   public boolean include(Entry<? extends TableModel, ? extends Integer>
      return !removedRowIndices.contains(entry.getIdentifier());
   }
};

// create menu

var menuBar = new JMenuBar();
setJMenuBar(menuBar);

var selectionMenu = new JMenu("Selection");
menuBar.add(selectionMenu);

rowsItem = new JCheckBoxMenuItem("Rows");
columnsItem = new JCheckBoxMenuItem("Columns");
cellsItem = new JCheckBoxMenuItem("Cells");

rowsItem.setSelected(table.getRowSelectionAllowed());
columnsItem.setSelected(table.getColumnSelectionAllowed());
cellsItem.setSelected(table.getCellSelectionEnabled());

rowsItem.addActionListener(_ -> {
   table.clearSelection();
   table.setRowSelectionAllowed(rowsItem.isSelected());
   updateCheckboxMenuItems();
});
selectionMenu.add(rowsItem);

columnsItem.addActionListener(_ -> {
   table.clearSelection();
   table.setColumnSelectionAllowed(columnsItem.isSelected());
   updateCheckboxMenuItems();
});
selectionMenu.add(columnsItem);

cellsItem.addActionListener(_ -> {
   table.clearSelection();
   table.setCellSelectionEnabled(cellsItem.isSelected());
   updateCheckboxMenuItems();
```

```
});
selectionMenu.add(cellsItem);

var tableMenu = new JMenu("Edit");
menuBar.add(tableMenu);

var hideColumnsItem = new JMenuItem("Hide Columns");
hideColumnsItem.addActionListener(_ -> {
   int[] selected = table.getSelectedColumns();
   TableColumnModel columnModel = table.getColumnModel();

   // remove columns from view, starting at the last
   // index so that column numbers aren't affected

   for (int i = selected.length - 1; i >= 0; i--) {
      TableColumn column = columnModel.getColumn(selected[i]);
      table.removeColumn(column);

      // store removed columns for "show columns" command

      removedColumns.add(column);
   }
});
tableMenu.add(hideColumnsItem);

var showColumnsItem = new JMenuItem("Show Columns");
showColumnsItem.addActionListener(_ -> {
   // restore all removed columns
   for (TableColumn tc : removedColumns)
      table.addColumn(tc);
   removedColumns.clear();
});
tableMenu.add(showColumnsItem);

var hideRowsItem = new JMenuItem("Hide Rows");
hideRowsItem.addActionListener(_ -> {
   int[] selected = table.getSelectedRows();
   for (int i : selected)
      removedRowIndices.add(table.convertRowIndexToModel(i));
   sorter.setRowFilter(filter);
});
tableMenu.add(hideRowsItem);

var showRowsItem = new JMenuItem("Show Rows");
```

```
        showRowsItem.addActionListener(_ -> {
           removedRowIndices.clear();
           sorter.setRowFilter(filter);
        });
        tableMenu.add(showRowsItem);

        var printSelectionItem = new JMenuItem("Print Selection");
        printSelectionItem.addActionListener(_ -> {
           int[] selected = table.getSelectedRows();
           System.out.println("Selected rows: " + Arrays.toString(selected));
           selected = table.getSelectedColumns();
           System.out.println("Selected columns: " + Arrays.toString(selected));
        });
        tableMenu.add(printSelectionItem);
     }

     private void updateCheckboxMenuItems() {
        rowsItem.setSelected(table.getRowSelectionAllowed());
        columnsItem.setSelected(table.getColumnSelectionAllowed());
        cellsItem.setSelected(table.getCellSelectionEnabled());
     }
}
```

javax.swing.table.TableModel 1.2

- Class getColumnClass(int columnIndex)
  gets the class for the values in this column. This information is used for sorting and rendering.

javax.swing.JTable 1.2

- TableColumnModel getColumnModel()
  gets the column model that describes the arrangement of the table columns.
- void setAutoResizeMode(int mode)
  sets the mode for automatic resizing of table columns. . The mode parameter is one of AUTO_RESIZE_OFF, AUTO_RESIZE_NEXT_COLUMN, AUTO_RESIZE_SUBSEQUENT_COLUMNS, AUTO_RESIZE_LAST_COLUMN, and AUTO_RESIZE_ALL_COLUMNS

- int getRowMargin()
- void setRowMargin(int margin)
  get or set the amount of empty space between cells in adjacent rows.
- int getRowHeight()
- void setRowHeight(int height)
  get or set the default height of all rows of the table.
- int getRowHeight(int row)
- void setRowHeight(int row, int height)
  get or set the height of the given row of the table.
- ListSelectionModel getSelectionModel()
  returns the list selection model. You need that model to choose between row, column, and cell selection.
- boolean getRowSelectionAllowed()
- void setRowSelectionAllowed(boolean b)
  get or set the rowSelectionAllowed property. If true, rows are selected when the user clicks on cells.
- boolean getColumnSelectionAllowed()
- void setColumnSelectionAllowed(boolean b)
  get or set the columnSelectionAllowed property. If true, columns are selected when the user clicks on cells.
- boolean getCellSelectionEnabled()
  returns true if both rowSelectionAllowed and columnSelectionAllowed are true.
- void setCellSelectionEnabled(boolean b)
  sets both rowSelectionAllowed and columnSelectionAllowed to b.
- void addColumn(TableColumn column)
  adds a column as the last column of the table view.
- void moveColumn(int from, int to)
  moves the column whose table index is from so that its index becomes to. Only the view is affected.
- void removeColumn(TableColumn column)
  removes the given column from the view.
- int convertRowIndexToModel(int index)6
- int convertColumnIndexToModel(int index)
  return the model index of the row or column with the given index. This value is different from index when rows are sorted or filtered, or when columns are moved or removed.

- void setRowSorter(RowSorter<? extends TableModel> sorter)
  sets the row sorter.

javax.swing.table.TableColumnModel1.2

- TableColumn getColumn(int index)
  gets the table column object that describes the column with the given view index.

javax.swing.table.TableColumn1.2

- TableColumn(int modelColumnIndex)
  constructs a table column for viewing the model column with the given index.
- void setPreferredWidth(int width)
- void setMinWidth(int width)
- void setMaxWidth(int width)
  set the preferred, minimum, and maximum width of this table column to width.
- void setWidth(int width)
  sets the actual width of this column to width.
- void setResizable(boolean b)
  If b is true, this column is resizable.

javax.swing.ListSelectionModel1.2

- void setSelectionMode(int mode)
  sets the selection mode to one of SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, and MULTIPLE_INTERVAL_SELECTION

javax.swing.DefaultRowSorter<M, I>6

- void setComparator(int column, Comparator<?>

comparator)
sets the comparator to be used with the given column.

- void setSortable(int column, boolean enabled)
  enables or disables sorting for the given column.
- void setRowFilter(RowFilter<? super M,? super I> filter)
  sets the row filter.

javax.swing.table.TableRowSorter<M extends TableModel>6

- void setStringConverter(TableStringConverter stringConverter)
  sets the string converter used for sorting and filtering.

javax.swing.table.TableStringConverter6

- abstract String toString(TableModel model, int row, int column)
  converts the model value at the given location to a string; you can override this method.

javax.swing.RowFilter<M, I>6

- boolean include(RowFilter.Entry<? extends M,? extends I> entry)
  specifies the rows that are retained; you can override this method.
- static <M,I> RowFilter<M,I> numberFilter(RowFilter.ComparisonType type, Number number, int... indices)
- static <M,I> RowFilter<M,I> dateFilter(RowFilter.ComparisonType type, Date date, int... indices)
  return a filter that includes rows containing values that match the given comparison to the given number or date. The comparison type is one of EQUAL, NOT_EQUAL, AFTER, or BEFORE. If any column model indexes are

given, only those columns are searched; otherwise, all columns are searched. For the number filter, the class of the cell value must match the class of number.

- static <M,I> RowFilter<M,I> regexFilter(String regex, int... indices)
  returns a filter that includes rows that have a string value matching the given regular expression. If any column model indexes are given, only those columns are searched; otherwise, all columns are searched. Note that the string returned by the getStringValue method of RowFilter.Entry is matched.
- static <M,I> RowFilter<M,I> andFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)
- static <M,I> RowFilter<M,I> orFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)
  return a filter that includes the entries included by all filters, or at least one of the filters.
- static <M,I> RowFilter<M,I> notFilter(RowFilter<M,I> filter)
  returns a filter that includes the entries not included by the given filter.

javax.swing.RowFilter.Entry<M, I>6

- I getIdentifier()
  returns the identifier of this row entry.
- M getModel()
  returns the model of this row entry.
- Object getValue(int index)
  returns the value stored at the given index of this row.
- int getValueCount()
  returns the number of values stored in this row.
- String getStringValue(int index)
  returns the value stored at the given index of this row, converted to a string. The TableRowSorter produces entries whose getStringValue calls the sorter's string converter.

12.3. Cell Rendering and Editing

As you saw in Section 12.2.2, the column type determines how the cells are rendered. There are default renderers for the types Boolean and Icon that render a checkbox or icon. For all other types, you need to install a custom renderer.

12.3.1. Rendering Cells

Table cell renderers are similar to the list cell renderers that you saw earlier. They implement the TableCellRenderer interface that has a single method:

```
Component getTableCellRendererComponent(JTable table, Object va
    boolean isSelected, boolean hasFocus, int row, int column)
```

That method is called when the table needs to draw a cell. You return a component whose paint method is then invoked to fill the cell area.

The table in Figure 12.8 contains cells of type Color. The renderer simply returns a panel with a background color that is the color object stored in the cell. The color is passed as the value parameter.

```
class ColorTableCellRenderer extends JPanel implements TableCellR
   public Component getTableCellRendererComponent(JTable table,
         boolean isSelected, boolean hasFocus, int row, int column) {
      setBackground((Color) value);
      if (hasFocus)
         setBorder(UIManager.getBorder("Table.focusCellHighlightBor
      else
         setBorder(null);
      return this;
   }
}
```

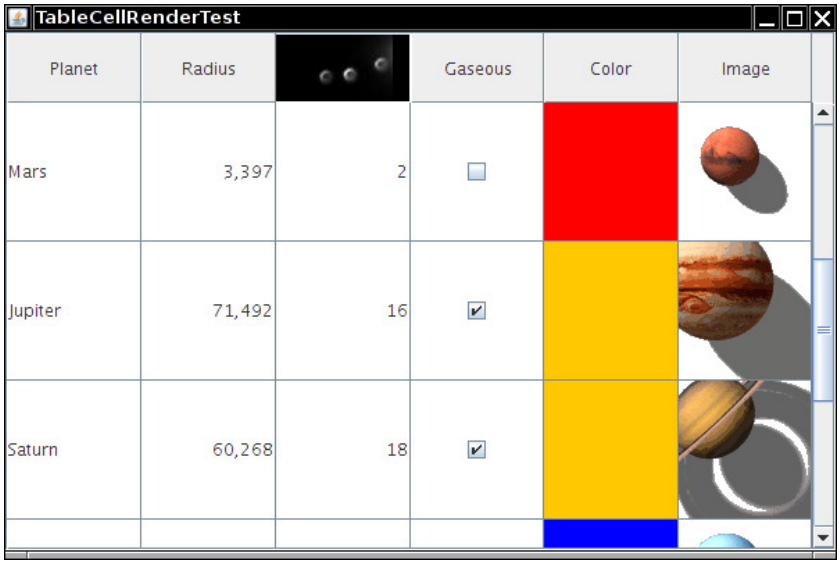


Figure 12.8: A table with cell renderers

As you can see, the renderer draws a border when the cell has focus. (We ask the UIManager for the correct border. To find the lookup key, we peeked into the source code of the DefaultTableCellRenderer class.)

> If your renderer simply draws a text string or an icon, you can extend the DefaultTableCellRenderer class. It takes care of rendering the focus and selection status for you.

You need to tell the table to use this renderer with all objects of type Color. The setDefaultRenderer method of the JTable class lets you establish this association. Supply a Class object and the renderer:

```
table.setDefaultRenderer(Color.class, new ColorTableCellRenderer())
```

That renderer is now used for all objects of the given type in this table.

If you want to select a renderer based on some other criterion,

you need to subclass the JTable class and override the getCellRenderer method.

12.3.2. Rendering the Header

To display an icon in the header, set the header value:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

However, the table header isn't smart enough to choose an appropriate renderer for the header value. You have to install the renderer manually. For example, to show an image icon in a column header, call

```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIc
```

12.3.3. Editing Cells

To enable cell editing, the table model must indicate which cells are editable by defining the isCellEditable method. Most commonly, you will want to make certain columns editable. In the example program, we allow editing in four columns.

```
public boolean isCellEditable(int r, int c) {
   return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GA
      || c == COLOR_COLUMN;
}
```

The AbstractTableModel defines the isCellEditable method to always return false. The DefaultTableModel overrides the method to always return true.

If you run the sample program at the end of this section, note that you can click the checkboxes in the Gaseous column and turn the check marks on and off. If you click a cell in the Moons column, a combo box appears (see Figure 12.9). You will shortly see how to install such a combo box as a cell editor.

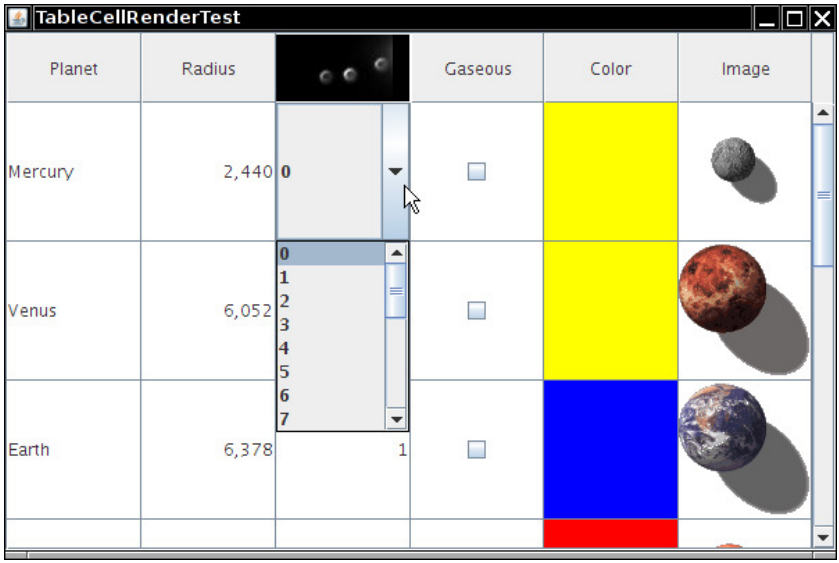


Figure 12.9: A cell editor

Finally, click a cell in the first column. The cell gains focus. You can start typing, and the cell contents change.

What you just saw in action are the three variations of the DefaultCellEditor class. A DefaultCellEditor can be constructed with a JTextField, a JCheckBox, or a JComboBox. The JTable class automatically installs a checkbox editor for Boolean cells and a text field editor for all editable cells that don't supply their own renderer. The text fields let the user edit the strings that result from applying toString to the return value of the getValueAt method of the table model.

When the edit is complete, the edited value is retrieved by calling the getCellEditorValue method of your editor. That method should return a value of the correct type (that is, the type returned by the getColumnType method of the model).

To get a combo box editor, set a cell editor manually—the JTable component has no idea what values might be appropriate for a particular type. For the Moons column, we wanted to enable the user to pick any value between 0 and 20. Here is the code for initializing the combo box:

```
var moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
   moonCombo.addItem(i);
```

To construct a DefaultCellEditor, supply the combo box in the constructor:

```
var moonEditor = new DefaultCellEditor(moonCombo);
```

Next, we need to install the editor. Unlike the color cell renderer, this editor does not depend on the object type—we don't necessarily want to use it for all objects of type Integer. Instead, we need to install it into a particular column:

```
moonColumn.setCellEditor(moonEditor);
```

12.3.4. Custom Editors

Run the example program again and click a color. A color chooser pops up and lets you pick a new color for the planet. Select a color and click OK. The cell color is updated (see Figure 12.10).

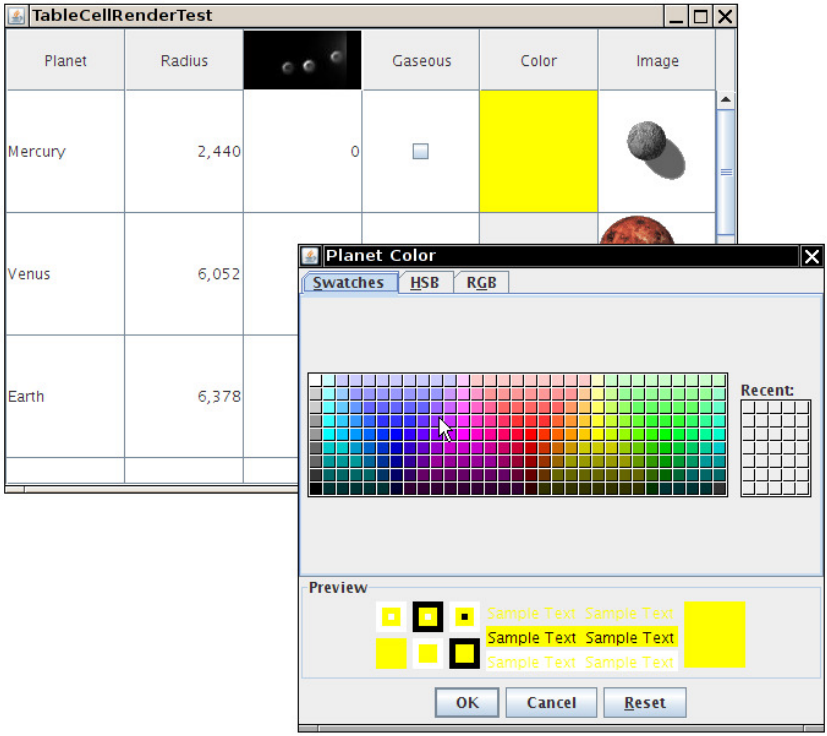


Figure 12.10: Editing the cell color with a color chooser

The color cell editor is not a standard table cell editor but a custom implementation. To create a custom cell editor, implement the TableCellEditor interface. That interface is a bit tedious, and an AbstractCellEditor class is provided to take care of the event handling details.

The getTableCellEditorComponent method of the TableCellEditor interface requests a component to render the cell. It is exactly the same as the getTableCellRendererComponent method of the TableCellRenderer interface, except that there is no focus parameter. When the cell is being edited, it is presumed to have focus. The editor component temporarily replaces the renderer when the editing is in progress. In our example, we return a blank panel that is not colored. This is an indication to the user that the cell is currently being edited.

Next, you want to have your editor pop up when the user clicks on the cell.

The JTable class calls your editor with an event (such as a mouse click) to find out if that event is acceptable to initiate the editing process. The AbstractCellEditor class defines the method to accept all events.

```
public boolean isCellEditable(EventObject anEvent) {
    return true;
}
```

However, if you override this method to return false, the table would not go through the trouble of inserting the editor component.

Once the editor component is installed, the shouldSelectCell method is called, presumably with the same event. You should initiate editing in this method—for example, by popping up an external edit dialog box.

```
public boolean shouldSelectCell(EventObject anEvent) {
   colorDialog.setVisible(true);
   return true;
}
```

If the user cancels the edit, the table calls the cancelCellEditing method. If the user has clicked on another table cell, the table calls the stopCellEditing method. In both cases, you should hide the dialog box. When your stopCellEditing method is called, the table would like to use the partially edited value. You should return true if the current value is valid. In the color chooser, any value is valid. But if you edit other data, you can ensure that only valid data are retrieved from the editor.

Also, you should call the superclass methods that take care of event firing—otherwise, the editing won't be properly canceled.

```
public void cancelCellEditing() {
    colorDialog.setVisible(false);
```

```
        super.cancelCellEditing();
    }
```

Finally, you need a method that yields the value that the user supplied in the editing process:

```
    public Object getCellEditorValue() {
        return colorChooser.getColor();
    }
```

To summarize, your custom editor should do the following:

1. Extend the AbstractCellEditor class and implement the TableCellEditor interface.
2. Define the getTableCellEditorComponent method to supply a component. This can either be a dummy component (if you pop up a dialog box) or a component for in-place editing such as a combo box or text field.
3. Define the shouldSelectCell, stopCellEditing, and cancelCellEditing methods to handle the start, completion, and cancellation of the editing process. The stopCellEditing and cancelCellEditing methods should call the superclass methods to ensure that listeners are notified.
4. Define the getCellEditorValue method to return the value that is the result of the editing process.

Finally, indicate when the user is finished editing by calling the stopCellEditing and cancelCellEditing methods. When constructing the color dialog box, we install the accept and cancel callbacks that fire these events.

```
    colorDialog = JColorChooser.createDialog(null, "Planet Color", false, c
        EventHandler.create(ActionListener.class, this, "stopCellEditing"),
        EventHandler.create(ActionListener.class, this, "cancelCellEditing"
```

This completes the implementation of the custom editor.

You now know how to make a cell editable and how to install an editor. There is one remaining issue—how to update the model with the value that the user edited. When editing is

complete, the JTable class calls the following method of the table model:

```
void setValueAt(Object value, int r, int c)
```

You need to override the method to store the new value. The value parameter is the object that was returned by the cell editor. If you implemented the cell editor, you know the type of the object you return from the getCellEditorValue method. In the case of the DefaultCellEditor, there are three possibilities for that value. It is a Boolean if the cell editor is a checkbox, a string if it is a text field, and, if the value comes from a combo box, it is the object that the user selected.

If the value object does not have the appropriate type, you need to convert it. That happens most commonly when a number is edited in a text field. In our example, we populated the combo box with Integer objects so that no conversion is necessary.

Listing 12.4 **v2ch12/tableCellRender/TableCellRenderFrame.java**

```
package v2ch12.tableCellRender;

import module java.desktop;

/**
 * This frame contains a table of planet data.
 */
public class TableCellRenderFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 600;
   private static final int DEFAULT_HEIGHT = 400;

   public TableCellRenderFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      var model = new PlanetTableModel();
      var table = new JTable(model);
      table.setRowSelectionAllowed(false);

      // set up renderers and editors
```

```
      table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());
      table.setDefaultEditor(Color.class, new ColorTableCellEditor());

      var moonCombo = new JComboBox<Integer>();
      for (int i = 0; i <= 20; i++)
         moonCombo.addItem(i);

      TableColumnModel columnModel = table.getColumnModel();
      TableColumn moonColumn = columnModel.getColumn(PlanetTableModel
      moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));
      moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.cla
      moonColumn.setHeaderValue(new ImageIcon("images/Moons.gif"));

      // show table

      table.setRowHeight(100);
      add(new JScrollPane(table), BorderLayout.CENTER);
   }
}
```

Listing 12.5 **v2ch12/tableCellRender/PlanetTableModel.java**

```
package v2ch12.tableCellRender;

import module java.desktop;

/**
 * The planet table model specifies the values, rendering and editing propertie
 * data.
 */
public class PlanetTableModel extends AbstractTableModel {
   public static final int PLANET_COLUMN = 0;
   public static final int MOONS_COLUMN = 2;
   public static final int GASEOUS_COLUMN = 3;
   public static final int COLOR_COLUMN = 4;

   private Object[][] cells = {
         { "Mercury", 2440.0, 0, false, Color.YELLOW,
               new ImageIcon("images/Mercury.gif") },
         { "Venus", 6052.0, 0, false, Color.YELLOW,
               new ImageIcon("images/Venus.gif") },
         { "Earth", 6378.0, 1, false, Color.BLUE,
```

```
            new ImageIcon("images/Earth.gif") },
        { "Mars", 3397.0, 2, false, Color.RED,
            new ImageIcon("images/Mars.gif") },
        { "Jupiter", 71492.0, 16, true, Color.ORANGE,
            new ImageIcon("images/Jupiter.gif") },
        { "Saturn", 60268.0, 18, true, Color.ORANGE,
            new ImageIcon("images/Saturn.gif") },
        { "Uranus", 25559.0, 17, true, Color.BLUE,
            new ImageIcon("images/Uranus.gif") },
        { "Neptune", 24766.0, 8, true, Color.BLUE,
            new ImageIcon("images/Neptune.gif") },
        { "Pluto", 1137.0, 1, false, Color.BLACK,
            new ImageIcon("images/Pluto.gif") } };

private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Co
        "Image" };

public String getColumnName(int c) {
    return columnNames[c];
}

public Class<?> getColumnClass(int c) {
    return cells[0][c].getClass();
}

public int getColumnCount() {
    return cells[0].length;
}

public int getRowCount() {
    return cells.length;
}

public Object getValueAt(int r, int c) {
    return cells[r][c];
}

public void setValueAt(Object obj, int r, int c) {
    cells[r][c] = obj;
}

public boolean isCellEditable(int r, int c) {
    return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GASEOUS
            || c == COLOR_COLUMN;
```

```
   }
}
```

Listing 12.6 **v2ch12/tableCellRender/ColorTableCellRenderer.java**

```
package v2ch12.tableCellRender;

import module java.desktop;

/**
 * This renderer renders a color value as a panel with the given color.
 */
public class ColorTableCellRenderer extends JPanel implements TableCellRend
   public Component getTableCellRendererComponent(JTable table, Object val
         boolean isSelected, boolean hasFocus, int row, int column) {
      setBackground((Color) value);
      if (hasFocus)
         setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
      else
         setBorder(null);
      return this;
   }
}
```

Listing 12.7 **v2ch12/tableCellRender/ColorTableCellEditor.java**

```
package v2ch12.tableCellRender;

import module java.base;
import module java.desktop;

/**
 * This editor pops up a color dialog to edit a cell value.
 */
public class ColorTableCellEditor extends AbstractCellEditor implements Table
   private JColorChooser colorChooser;
   private JDialog colorDialog;
   private JPanel panel;
```

```
public ColorTableCellEditor() {
   panel = new JPanel();
   // prepare color dialog

   colorChooser = new JColorChooser();
   colorDialog = JColorChooser.createDialog(null, "Planet Color", false, colorC
         EventHandler.create(ActionListener.class, this, "stopCellEditing"),
         EventHandler.create(ActionListener.class, this, "cancelCellEditing"));
}

public Component getTableCellEditorComponent(JTable table, Object value,
      boolean isSelected, int row, int column) {
   // this is where we get the current Color value. We store it in the dialog in
   // user starts editing
   colorChooser.setColor((Color) value);
   return panel;
}

public boolean shouldSelectCell(EventObject anEvent) {
   // start editing
   colorDialog.setVisible(true);

   // tell caller it is ok to select this cell
   return true;
}

public void cancelCellEditing() {
   // editing is canceled--hide dialog
   colorDialog.setVisible(false);
   super.cancelCellEditing();
}

public boolean stopCellEditing() {
   // editing is complete--hide dialog
   colorDialog.setVisible(false);
   super.stopCellEditing();

   // tell caller it is ok to use color value
   return true;
}

public Object getCellEditorValue() {
   return colorChooser.getColor();
}
```

}

javax.swing.JTable1.2

- TableCellRenderer getDefaultRenderer(Class<?> type)
  gets the default renderer for the given type.
- TableCellEditor getDefaultEditor(Class<?> type)
  gets the default editor for the given type.

javax.swing.table.TableCellRenderer1.2

- Component getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)
  returns a component whose paint method is invoked to render a table cell.

javax.swing.table.TableColumn1.2

- void setCellEditor(TableCellEditor editor)
- void setCellRenderer(TableCellRenderer renderer)
  set the cell editor or renderer for all cells in this column.
- void setHeaderRenderer(TableCellRenderer renderer)
  sets the cell renderer for the header cell in this column.
- void setHeaderValue(Object value)
  sets the value to be displayed for the header in this column.

javax.swing.DefaultCellEditor1.2

- DefaultCellEditor(JComboBox comboBox)
  constructs a cell editor that presents the combo box for selecting cell values.

javax.swing.table.TableCellEditor1.2

- Component getTableCellEditorComponent(JTable table, Object value, boolean selected, int row, int column)
  returns a component whose paint method renders a table cell.

javax.swing.CellEditor1.2

- boolean isCellEditable(EventObject event)
  returns true if the event is suitable for initiating the editing process for this cell.
- boolean shouldSelectCell(EventObject anEvent)
  starts the editing process. Returns true if the edited cell should be selected. Normally, you want to return true, but you can return false if you don't want the editing process to change the cell selection.
- void cancelCellEditing()
  cancels the editing process. You can abandon partial edits.
- boolean stopCellEditing()
  stops the editing process, with the intent of using the result. Returns true if the edited value is in a proper state for retrieval.
- Object getCellEditorValue()
  returns the edited result.
- void addCellEditorListener(CellEditorListener l)
- void removeCellEditorListener(CellEditorListener l)
  add or remove the obligatory cell editor listener.

12.4. Trees

Every computer user who has worked with a hierarchical file system has seen tree displays. Of course, directories and files are only one of the many examples of tree-like organizations. Many tree structures arise in everyday life, such as the hierarchy of countries, states, and cities shown in Figure 12.11.

.

Figure 12.11: A hierarchy of countries, states, and cities

As programmers, we often need to display tree structures. Fortunately, the Swing library has a JTree class for this purpose. The JTree class (together with its helper classes) takes care of laying out the tree and processing user requests for expanding and collapsing nodes. In this section, you will learn how to put the JTree class to use.

As with the other complex Swing components, we must focus on the common and useful cases and cannot cover every nuance. If you want to achieve something unusual, we recommend that you consult Graphic Java™, Third Edition, by David M. Geary or Core Swing by Kim Topley.

Before going any further, let's settle on some terminology (see Figure 12.12). A tree is composed of nodes. Every node is either a leaf or it has child nodes. Every node, with the exception of the root node, has exactly one parent. A tree has exactly one root node. Sometimes you have a collection of trees, each with its own root node. Such a collection is called a forest.

.

Figure 12.12: Tree terminology

12.4.1. Simple Trees

In our first example program, we will simply display a tree with a few nodes (see Figure 12.14). As with many other Swing components, you need to provide a model of the data, and the component displays it for you. To construct a JTree, supply the tree model in the constructor:

```
TreeModel model = . . .;
var tree = new JTree(model);
```

There are also constructors that construct trees out of a

collection of elements:

```
JTree(Object[] nodes)
JTree(Vector<?> nodes)
JTree(Hashtable<?, ?> nodes) // the values become the nodes
```

These constructors are not very useful. They merely build a forest of trees, each with a single node. The third constructor seems particularly useless because the nodes appear in the seemingly random order determined by the hash codes of the keys.

How do you obtain a tree model? You can construct your own model by creating a class that implements the TreeModel interface. You will see later in this chapter how to do that. For now, we will stick with the DefaultTreeModel that the Swing library supplies.

To construct a default tree model, you must supply a root node.

```
TreeNode root = . . .;
var model = new DefaultTreeModel(root);
```

TreeNode is another interface. Populate the default tree model with objects of any class that implements the interface. For now, we will use the concrete node class that Swing supplies—namely, DefaultMutableTreeNode. This class implements the MutableTreeNode interface, a subinterface of TreeNode (see Figure 12.13).

.

Figure 12.13: Tree classes

A default mutable tree node holds an object—the user object. The tree renders the user objects for all nodes. Unless you specify a renderer, the tree displays the string that is the result of the toString method.

In our first example, we use strings as user objects. In practice,

you would usually populate a tree with more expressive user objects. For example, when displaying a directory tree, it makes sense to use File objects for the nodes.

You can specify the user object in the constructor, or you can set it later with the setUserObject method.

```
var node = new DefaultMutableTreeNode("Texas");
. . .
node.setUserObject("California");
```

Next, you need to establish the parent/child relationships between the nodes. Start with the root node and use the add method to add the children:

```
var root = new DefaultMutableTreeNode("World");
var country = new DefaultMutableTreeNode("USA");
root.add(country);
var state = new DefaultMutableTreeNode("California");
country.add(state);
```

Figure 12.14 illustrates how the tree will look.



Figure 12.14: A simple tree

Link up all nodes in this fashion. Then, construct a DefaultTreeModel with the root node. Finally, construct a JTree with the tree model.

```
var treeModel = new DefaultTreeModel(root);
var tree = new JTree(treeModel);
```

Or, as a shortcut, you can simply pass the root node to the JTree constructor. Then the tree automatically constructs a default tree model:

```
var tree = new JTree(root);
```

Listing 12.8 contains the complete code.

Listing 12.8 **v2ch12/tree/SimpleTreeFrame.java**

```
package v2ch12.tree;

import module java.desktop;

/**
 * This frame contains a simple tree that displays a manually constructed tree
 */
class SimpleTreeFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public SimpleTreeFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // set up tree model data

      var root = new DefaultMutableTreeNode("World");
      var country = new DefaultMutableTreeNode("USA");
      root.add(country);
      var state = new DefaultMutableTreeNode("California");
      country.add(state);
      var city = new DefaultMutableTreeNode("San Jose");
      state.add(city);
      city = new DefaultMutableTreeNode("Cupertino");
```

```
        state.add(city);
        state = new DefaultMutableTreeNode("Michigan");
        country.add(state);
        city = new DefaultMutableTreeNode("Ann Arbor");
        state.add(city);
        country = new DefaultMutableTreeNode("Germany");
        root.add(country);
        state = new DefaultMutableTreeNode("Schleswig-Holstein");
        country.add(state);
        city = new DefaultMutableTreeNode("Kiel");
        state.add(city);

        // construct tree and put it in a scroll pane

        var tree = new JTree(root);
        add(new JScrollPane(tree));
    }
}
```

When you run the program, the tree first looks as in Figure 12.15. Only the root node and its children are visible. Click on the circle icons (the handles) to open up the subtrees. The line sticking out from the handle icon points to the right when the subtree is collapsed and down when the subtree is expanded (see Figure 12.16). We don't know what the designers of the Metal look-and-feel had in mind, but we think of the icon as a door handle. You push down on the handle to open the subtree.



Figure 12.15: The initial tree display

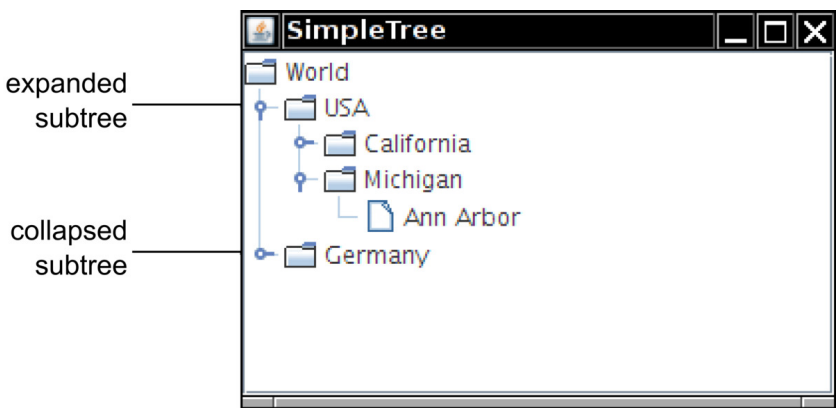


Figure 12.16: Collapsed and expanded subtrees

Of course, the display of the tree depends on the selected look-and-feel. We just described the Metal look-and-feel. In the Windows look-and-feel, the handles have the more familiar look—a "–" or "+" in a box (see Figure 12.17).

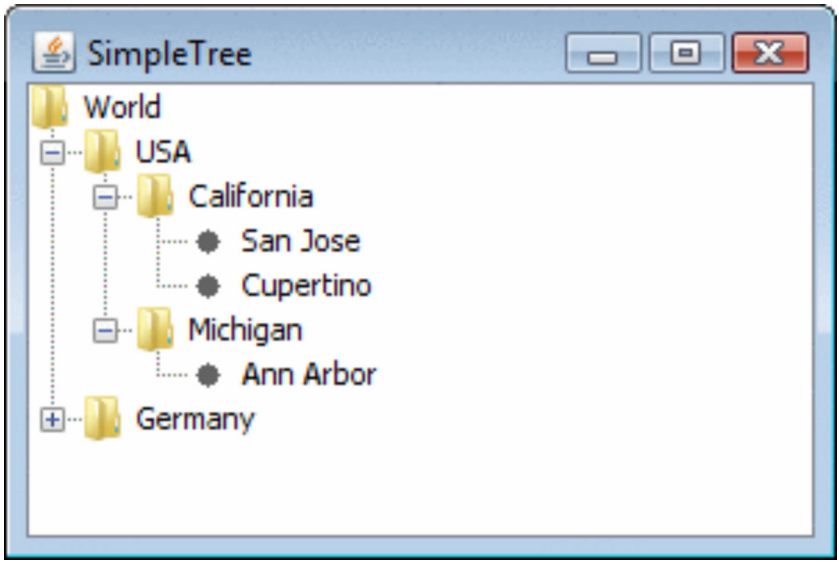


Figure 12.17: A tree with the Windows look-and-feel

You can use the following magic incantation to turn off the lines joining parents and children (see Figure 12.18):

```
tree.putClientProperty("JTree.lineStyle", "None");
```



Figure 12.18: A tree with no connecting lines

Conversely, to make sure that the lines are shown, use

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

Another line style, "Horizontal", is shown in Figure 12.19. The tree is displayed with horizontal lines separating only the children of the root. I'm not quite sure what it is good for.

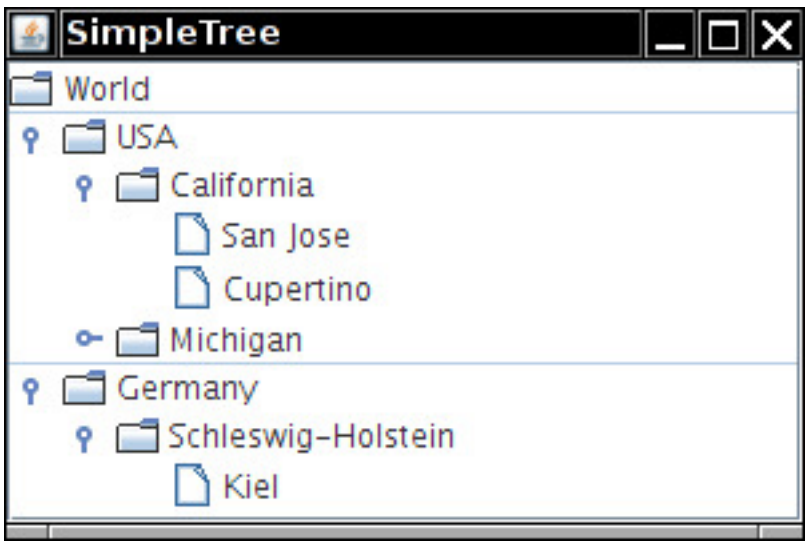


Figure 12.19: A tree with the horizontal line style

By default, there is no handle for collapsing the root of the tree. If you like, you can add one with the call

```
tree.setShowsRootHandles(true);
```

Figure 12.20 shows the result. Now you can collapse the entire tree into the root node.

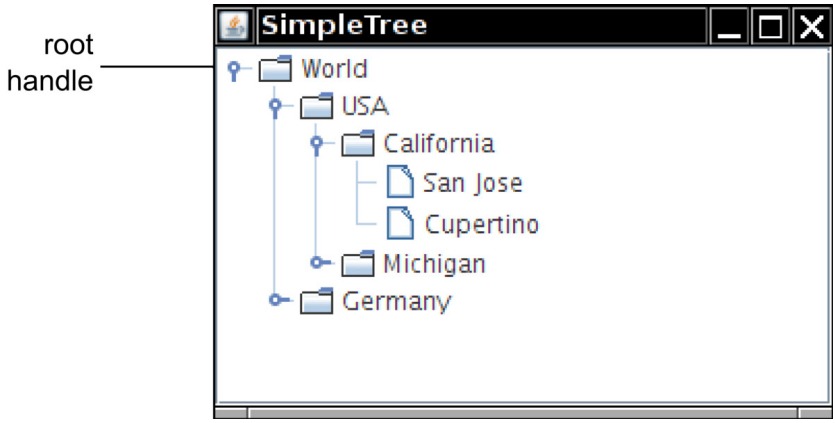


Figure 12.20: A tree with a root handle

Conversely, you can hide the root altogether. You will thus display a forest—a set of trees, each with its own root. You still must join all trees in the forest to a common root; then, hide the root with the instruction

```
tree.setRootVisible(false);
```

Look at Figure 12.21. There appear to be two roots, labeled "USA" and "Germany." The actual root that joins the two is made invisible.



Figure 12.21: A forest

Let's turn from the root to the leaves of the tree. Note that the leaves have an icon different from the other nodes (see Figure 12.22).

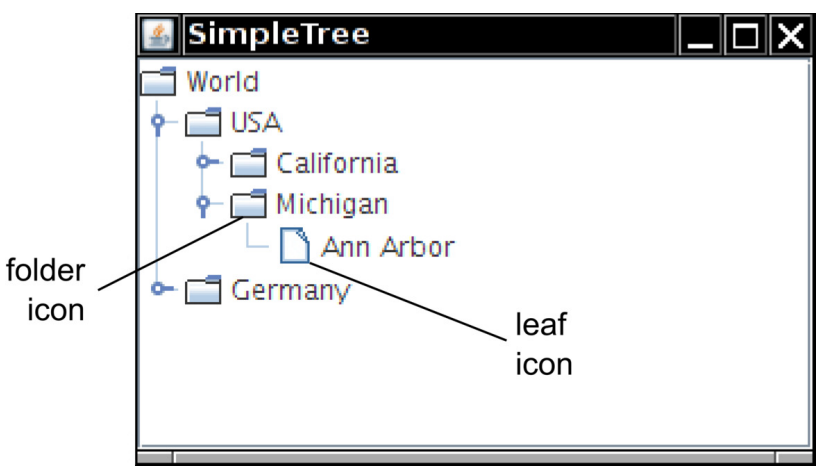


Figure 12.22: Leaf and folder icons

When the tree is displayed, each node is drawn with an icon.

There are actually three kinds of icons: a leaf icon, an opened nonleaf icon, and a closed nonleaf icon. For simplicity, we refer to the last two as folder icons.

The node renderer needs to know which icon to use for each node. By default, the decision process works like this: If the isLeaf method of a node returns true, then the leaf icon is used; otherwise, a folder icon is used.

The isLeaf method of the DefaultMutableTreeNode class returns true if the node has no children. Thus, nodes with children get folder icons, and nodes without children get leaf icons.

Sometimes, that behavior is not appropriate. Suppose we added a node "Montana" to our sample tree, but we're at a loss as to what cities to add. We would not want the state node to get a leaf icon because, conceptually, only the cities are leaves.

The JTree class has no idea which nodes should be leaves. It asks the tree model. If a childless node isn't automatically a conceptual leaf, you can ask the tree model to use a different criterion for leafiness—namely, to query the "allows children" node property.

For those nodes that should not have children, call

```
node.setAllowsChildren(false);
```

Then, tell the tree model to ask the value of the "allows children" property to determine whether a node should be displayed with a leaf icon. Use the setAsksAllowsChildren method of the DefaultTreeModel class to set this behavior:

```
model.setAsksAllowsChildren(true);
```

With this decision criterion, nodes that allow children get folder icons, and nodes that don't allow children get leaf icons.

Alternatively, if you construct the tree from the root node, supply the setting for the "asks allows children" property in the constructor.

```
var tree = new JTree(root, true); // nodes that don't allow children ge
```

javax.swing.JTree 1.2

- JTree(TreeModel model)
  constructs a tree from a tree model.
- JTree(TreeNode root)
- JTree(TreeNode root, boolean asksAllowChildren)
  construct a tree with a default tree model that displays the root and its children.
- void setShowsRootHandles(boolean b)
  If b is true, the root node has a handle for collapsing or expanding its children.
- void setRootVisible(boolean b)
  If b is true, then the root node is displayed. Otherwise, it is hidden.

javax.swing.tree.TreeNode 1.2

- boolean isLeaf()
  returns true if this node is conceptually a leaf.
- boolean getAllowsChildren()
  returns true if this node can have child nodes.

javax.swing.tree.MutableTreeNode 1.2

- void setUserObject(Object userObject)
  sets the “user object” that the tree node uses for rendering.

javax.swing.tree.TreeModel 1.2

- boolean isLeaf(Object node)
  returns true if node should be displayed as a leaf node.

javax.swing.tree.DefaultTreeModel1.2

- void setAsksAllowsChildren(boolean b)
  If b is true, nodes are displayed as leaves when their getAllowsChildren method returns false. Otherwise, they are displayed as leaves when their isLeaf method returns true.

javax.swing.tree.DefaultMutableTreeNode1.2

- DefaultMutableTreeNode(Object userObject)
  constructs a mutable tree node with the given user object.
- void add(MutableTreeNode child)
  adds a node as the last child of this node.
- void setAllowsChildren(boolean b)
  If b is true, children can be added to this node.

javax.swing.JComponent1.2

- void putClientProperty(Object key, Object value)
  adds a key/value pair to a small table that each component manages. This is an “escape hatch” mechanism that some Swing components use for storing properties specific to a look-and-feel.

12.4.2. Editing Trees and Tree Paths

In the next example program, you will see how to edit a tree. Figure 12.23 shows the user interface. If you click the Add Sibling or Add Child button, the program adds a new node (with title New) to the tree. If you click the Delete button, the program deletes the currently selected node.



Figure 12.23: Editing a tree

To implement this behavior, you need to find out which tree node is currently selected. The JTree class has a surprising way of identifying nodes in a tree. It does not deal with tree nodes but with paths of objects, called tree paths. A tree path starts at the root and consists of a sequence of child nodes (see Figure 12.24).

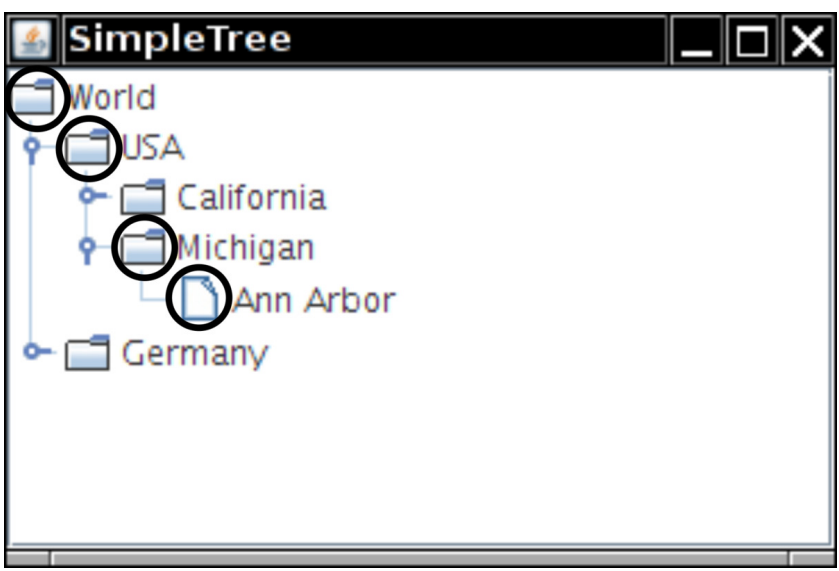


Figure 12.24: A tree path

You might wonder why the JTree class needs the whole path. Couldn't it just get a TreeNode and keep calling the getParent method? In fact, the JTree class knows nothing about the TreeNode interface. That interface is never used by the TreeModel interface; it is only used by the DefaultTreeModel implementation. You can have other tree models in which the nodes do not implement the TreeNode interface at all. If you use a tree model that manages other types of objects, those objects might not have getParent and getChild methods. They would of course need to have some other connection to each other. It is the job of the tree model to link nodes together. The JTree class itself has no clue about the nature of their linkage. For that reason, the JTree class always needs to work with complete paths.

The TreePath class manages a sequence of Object (not TreeNode!) references. A number of JTree methods return TreePath objects. When you have a tree path, you usually just need to know the terminal node, which you can get with the getLastPathComponent method. For example, to find out the currently selected node in a tree, use the getSelectionPath method of the JTree class. You will get a TreePath object back, from which you can retrieve the actual node.

```
TreePath selectionPath = tree.getSelectionPath();
var selectedNode = (DefaultMutableTreeNode) selectionPath.getLast
```

Actually, since this particular query is so common, there is a convenience method that gives the selected node immediately:

```
var selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedP
```

This method is not called getSelectedNode because the tree does not know that it contains nodes—its tree model deals only with paths of objects.

> Tree paths are one of the two ways in which the JTree class describes nodes. Quite a few JTree methods take or return an integer index—the row position. A row position is simply the row number (starting with 0) of the

node in the tree display. Only visible nodes have row numbers, and the row number of a node changes if other nodes before it are expanded, collapsed, or modified. For that reason, you should avoid row positions. All JTree methods that use row positions have equivalents that use tree paths instead.

Once you have the selected node, you can edit it. However, do not simply add children to a tree node:

```
selectedNode.add(newNode); // No!
```

If you change the structure of the nodes, you change the model but the associated view is not notified. You could send out a notification yourself, but if you use the insertNodeInto method of the DefaultTreeModel class, the model class takes care of that. For example, the following call appends a new node as the last child of the selected node and notifies the tree view:

```
model.insertNodeInto(newNode, selectedNode, selectedNode.getChil
```

The analogous call removeNodeFromParent removes a node and notifies the view:

```
model.removeNodeFromParent(selectedNode);
```

If you keep the node structure in place but change the user object, you should call the following method:

```
model.nodeChanged(changedNode);
```

The automatic notification is a major advantage of using the DefaultTreeModel. If you supply your own tree model, you have to implement automatic notification by hand. (See Core Swing by Kim Topley for details.)

The DefaultTreeModel class has a reload method that reloads the entire model. However, don't call reload simply to update the tree after making a few changes. When the tree is regenerated, all nodes beyond the root's

children are collapsed again. It will be quite disconcerting to your users if they have to keep expanding the tree after every change.

When the view is notified of a change in the node structure, it updates the display but does not automatically expand a node to show newly added children. In particular, if a user in our sample program adds a new child node to a node for which children are currently collapsed, the new node is silently added to the collapsed subtree. This gives the user no feedback that the command was actually carried out. In such a case, you should make a special effort to expand all parent nodes so that the newly added node becomes visible. Use the makeVisible method of the JTree class for this purpose. The makeVisible method expects a tree path leading to the node that should become visible.

Thus, you need to construct a tree path from the root to the newly inserted node. To get a tree path, first call the getPathToRoot method of the DefaultTreeModel class. It returns a TreeNode[] array of all nodes from a node to the root node. Pass that array to a TreePath constructor.

For example, here is how you make the new node visible:

```
TreeNode[] nodes = model.getPathToRoot(newNode);
var path = new TreePath(nodes);
tree.makeVisible(path);
```

It is curious that the DefaultTreeModel class feigns almost complete ignorance of the TreePath class, even though its job is to communicate with a JTree. The JTree class uses tree paths a lot, and it never uses arrays of node objects.

Now, suppose your tree is contained inside a scroll pane. After the tree node expansion, the new node might still not be visible because it falls outside the viewport. To overcome that problem, call

```
tree.scrollPathToVisible(path);
```

instead of calling makeVisible. This call expands all nodes along the path and tells the ambient scroll pane to scroll the node at the end of the path into view (see Figure 12.25).

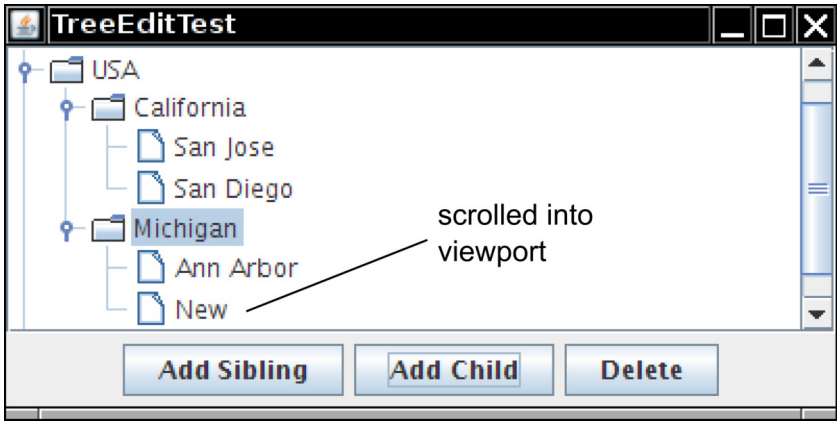


Figure 12.25: The scroll pane scrolls to display a new node.

By default, tree nodes cannot be edited. However, if you call

```
tree.setEditable(true);
```

the user can edit a node simply by double-clicking, editing the string, and pressing the Enter key. Double-clicking invokes the default cell editor, which is implemented by the DefaultCellEditor class (see Figure 12.26). It is possible to install other cell editors, using the same process that you have seen in our discussion of table cell editors.



Figure 12.26: The default cell editor

Listing 12.9 shows the complete source code of the tree editing program. Run the program, add a few nodes, and edit them by double-clicking. Observe how collapsed nodes expand to show added children and how the scroll pane keeps added nodes in the viewport.

Listing 12.9 **v2ch12/treeEdit/TreeEditFrame.java**

```
package v2ch12.treeEdit;

import module java.desktop;

/**
 * A frame with a tree and buttons to edit the tree.
 */
public class TreeEditFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 200;

   private DefaultTreeModel model;
   private JTree tree;

   public TreeEditFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
      // construct tree

      TreeNode root = makeSampleTree();
      model = new DefaultTreeModel(root);
      tree = new JTree(model);
      tree.setEditable(true);

      // add scroll pane with tree

      var scrollPane = new JScrollPane(tree);
      add(scrollPane, BorderLayout.CENTER);

      makeButtons();
   }

   public TreeNode makeSampleTree() {
      var root = new DefaultMutableTreeNode("World");
      var country = new DefaultMutableTreeNode("USA");
      root.add(country);
      var state = new DefaultMutableTreeNode("California");
      country.add(state);
      var city = new DefaultMutableTreeNode("San Jose");
      state.add(city);
      city = new DefaultMutableTreeNode("San Diego");
      state.add(city);
      state = new DefaultMutableTreeNode("Michigan");
      country.add(state);
      city = new DefaultMutableTreeNode("Ann Arbor");
      state.add(city);
      country = new DefaultMutableTreeNode("Germany");
      root.add(country);
      state = new DefaultMutableTreeNode("Schleswig-Holstein");
      country.add(state);
      city = new DefaultMutableTreeNode("Kiel");
      state.add(city);
      return root;
   }

   /**
    * Makes the buttons to add a sibling, add a child, and delete a node.
    */
   public void makeButtons() {
      var panel = new JPanel();
      var addSiblingButton = new JButton("Add Sibling");
```

```
addSiblingButton.addActionListener(_ -> {
   var selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPath

   if (selectedNode == null) return;

   var parent = (DefaultMutableTreeNode) selectedNode.getParent();

   if (parent == null) return;

   var newNode = new DefaultMutableTreeNode("New");

   int selectedIndex = parent.getIndex(selectedNode);
   model.insertNodeInto(newNode, parent, selectedIndex + 1);

   // now display new node

   TreeNode[] nodes = model.getPathToRoot(newNode);
   var path = new TreePath(nodes);
   tree.scrollPathToVisible(path);
});
panel.add(addSiblingButton);

var addChildButton = new JButton("Add Child");
addChildButton.addActionListener(_ -> {
   var selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPath

   if (selectedNode == null) return;

   var newNode = new DefaultMutableTreeNode("New");
   model.insertNodeInto(newNode, selectedNode, selectedNode.getChildC

   // now display new node

   TreeNode[] nodes = model.getPathToRoot(newNode);
   var path = new TreePath(nodes);
   tree.scrollPathToVisible(path);
});
panel.add(addChildButton);

var deleteButton = new JButton("Delete");
deleteButton.addActionListener(_ -> {
   var selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPath

   if (selectedNode != null && selectedNode.getParent() != null)
```

```
            model.removeNodeFromParent(selectedNode);
        });
        panel.add(deleteButton);
        add(panel, BorderLayout.SOUTH);
    }
}
```

javax.swing.JTree1.2

- TreePath getSelectionPath()
  gets the path to the currently selected node, or the path to the first selected node if multiple nodes are selected. Returns null if no node is selected.
- Object getLastSelectedPathComponent()
  gets the node object that represents the currently selected node, or the first node if multiple nodes are selected. Returns null if no node is selected.
- void makeVisible(TreePath path)
  expands all nodes along the path.
- void scrollPathToVisible(TreePath path)
  expands all nodes along the path and, if the tree is contained in a scroll pane, scrolls to ensure that the last node on the path is visible.

javax.swing.tree.TreePath1.2

- Object getLastPathComponent()
  gets the last object on this path—that is, the node object that the path represents.

javax.swing.tree.TreeNode1.2

- TreeNode getParent()
  returns the parent node of this node.
- TreeNode getChildAt(int index)
  looks up the child node at the given index. The index must be between 0 and getChildCount()– 1.
- int getChildCount()

returns the number of children of this node.

- Enumeration children()
  returns an enumeration object that iterates through all children of this node.

javax.swing.tree.DefaultTreeModel1.2

- void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)
  inserts newChild as a new child node of parent at the given index and notifies the tree model listeners.
- void removeNodeFromParent(MutableTreeNode node)
  removes node from this model and notifies the tree model listeners.
- void nodeChanged(TreeNode node)
  notifies the tree model listeners that node has changed.
- void nodesChanged(TreeNode parent, int[] changedChildIndexes)
  notifies the tree model listeners that all child nodes of parent with the given indexes have changed.
- void reload()
  reloads all nodes into the model. This is a drastic operation that you should use only if the nodes have changed completely because of some outside influence.

12.4.3. Node Enumeration

Sometimes you need to find a node in a tree by starting at the root and visiting all children until you have found a match. The DefaultMutableTreeNode class has several convenience methods for iterating through nodes.

The breadthFirstEnumeration and depthFirstEnumeration methods return enumeration objects whose nextElement method visits all children of the current node, using either a breadth-first or depth-first traversal. Figure 12.27 shows the traversals for a sample tree—the node labels indicate the order

in which the nodes are traversed.

.

Figure 12.27: Tree traversal orders

Breadth-first enumeration is the easiest to visualize. The tree is traversed in layers. The root is visited first, followed by all of its children, then the grandchildren, and so on.

To visualize depth-first enumeration, imagine a rat trapped in a tree-shaped maze. It rushes along the first path until it comes to a leaf. Then, it backtracks and turns around to the next path, and so on.

Computer scientists also call this postorder traversal because the search process visits the children before visiting the parents. The postOrderEnumeration method is a synonym for depthFirstEnumeration. For completeness, there is also a preOrderEnumeration, a depth-first search that enumerates parents before the children.

Here is the typical usage pattern:

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
   do something with breadthFirst.nextElement();
```

Finally, a related method, pathFromAncestorEnumeration, finds a path from an ancestor to a given node and enumerates the nodes along that path. That's no big deal—it just keeps calling getParent until the ancestor is found and then presents the path in reverse order.

In our next example program, we put node enumeration to work. The program displays inheritance trees of classes. Type the name of a class into the text field on the bottom of the frame. The class and all of its superclasses are added to the tree (see Figure 12.28).

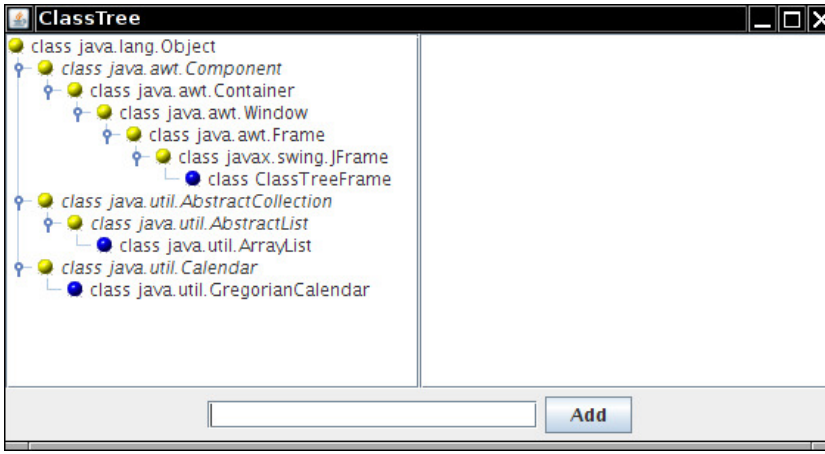


Figure 12.28: An inheritance tree

In this example, we take advantage of the fact that the user objects of the tree nodes can be objects of any type. Since our nodes describe classes, we store Class objects in the nodes.

We don't want to add the same class object twice, so we need to check whether a class already exists in the tree. The following method finds the node with a given user object if it exists in the tree:

```
public DefaultMutableTreeNode findUserObject(Object obj) {
   Enumeration e = root.breadthFirstEnumeration();
   while (e.hasMoreElements()) {
      DefaultMutableTreeNode node = (DefaultMutableTreeNode) e.n
      if (node.getUserObject().equals(obj))
         return node;
   }
   return null;
}
```

12.4.4. Rendering Nodes

In your applications, you will often need to change the way a tree component draws the nodes. The most common change is, of course, to choose different icons for nodes and leaves. Other

changes might involve changing the font of the node labels or drawing images at the nodes. All these changes are possible by installing a new tree cell renderer into the tree. By default, the JTree class uses DefaultTreeCellRenderer objects to draw each node. The DefaultTreeCellRenderer class extends the JLabel class. The label contains the node icon and the node label.

The cell renderer does not draw the "handles" for expanding and collapsing subtrees. The handles are part of the look-and-feel, and it is recommended that you do not change them.

You can customize the display in three ways.

- You can change the icons, font, and background color used by a DefaultTreeCellRenderer. These settings are used for all nodes in the tree.
- You can install a renderer that extends the DefaultTreeCellRenderer class and vary the icons, fonts, and background color for each node.
- You can install a renderer that implements the TreeCellRenderer interface to draw a custom image for each node.

Let us look at these possibilities one by one. The easiest customization is to construct a DefaultTreeCellRenderer object, change the icons, and install it into the tree:

```
var renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif")); // used for leaf
renderer.setClosedIcon(new ImageIcon("red-ball.gif")); // used for co
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif")); // used for
tree.setCellRenderer(renderer);
```

You can see the effect in Figure 12.28. We just use the "ball" icons as placeholders—presumably your user interface designer would supply you with appropriate icons to use for your applications.

We don't recommend that you change the font or background

color for an entire tree—that is really the job of the look-and-feel.

However, it can be useful to change the font of some nodes in a tree to highlight them. If you look carefully at Figure 12.28, you will notice that the abstract classes are set in italics.

To change the appearance of individual nodes, install a tree cell renderer. Tree cell renderers are very similar to the list cell renderers we discussed earlier in this chapter. The TreeCellRenderer interface has a single method:

```
Component getTreeCellRendererComponent(JTree tree, Object value
    boolean expanded, boolean leaf, int row, boolean hasFocus)
```

The getTreeCellRendererComponent method of the DefaultTreeCellRenderer class returns this—in other words, a label. (The DefaultTreeCellRenderer class extends the JLabel class.) To customize the component, extend the DefaultTreeCellRenderer class. Override the getTreeCellRendererComponent method as follows: Call the superclass method so it can prepare the label data, customize the label properties, and finally return this.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer {
   public Component getTreeCellRendererComponent(JTree tree, Obj
         boolean selected, boolean expanded, boolean leaf, int row, bo
      Component comp = super.getTreeCellRendererComponent(tree,
         expanded, leaf, row, hasFocus);
      DefaultMutableTreeNode node = (DefaultMutableTreeNode) val
      look at node.getUserObject();
      Font font = appropriate font;
      comp.setFont(font);
      return comp;
   }
}
```

The value parameter of the getTreeCellRendererComponent method is the node object, not the user object! Recall that the user object is a

feature of the DefaultMutableTreeNode, and that a JTree can contain nodes of an arbitrary type. If your tree uses DefaultMutableTreeNode nodes, you must retrieve the user object in a second step, as we did in the preceding code sample.

The DefaultTreeCellRenderer uses the same label object for all nodes, only changing the label text for each node. If you change the font for a particular node, you must set it back to its default value when the method is called again. Otherwise, all subsequent nodes will be drawn in the changed font! Look at the code in Listing 12.10 to see how to restore the font to the default.

The ClassNameTreeCellRenderer in Listing 12.10 sets the class name in either the normal or italic font, depending on the ABSTRACT modifier of the Class object. We don't want to set a particular font because we don't want to change whatever font the look-and-feel normally uses for labels. For that reason, we use the font from the label and derive an italic font from it. Recall that only a single shared JLabel object is returned by all calls. We need to hang on to the original font and restore it in the next call to the getTreeCellRendererComponent method.

Also, note how we change the node icons in the ClassTreeFrame constructor.

javax.swing.tree.DefaultMutableTreeNode1.2

- Enumeration breadthFirstEnumeration()
- Enumeration depthFirstEnumeration()
- Enumeration preOrderEnumeration()
- Enumeration postOrderEnumeration()

  return enumeration objects for visiting all nodes of the tree model in a particular order. In breadth-first traversal, children that are closer to the root are visited before those that are farther away. In depth-first traversal, all children of a node are completely enumerated before its siblings

are visited. The postOrderEnumeration method is a synonym for depthFirstEnumeration. The preorder traversal is identical to the postorder traversal except that parents are enumerated before their children.

javax.swing.tree.TreeCellRenderer1.2

- Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)
  returns a component whose paint method is invoked to render a tree cell.

javax.swing.tree.DefaultTreeCellRenderer1.2

- void setLeafIcon(Icon icon)
- void setOpenIcon(Icon icon)
- void setClosedIcon(Icon icon)
  set the icon to show for a leaf node, an expanded node, and a collapsed node.

12.4.5. Listening to Tree Events

Most commonly, a tree component is paired with some other component. When the user selects tree nodes, some information shows up in another window. See Figure 12.29 for an example. When the user selects a class, the instance and static variables of that class are displayed in the text area to the right.

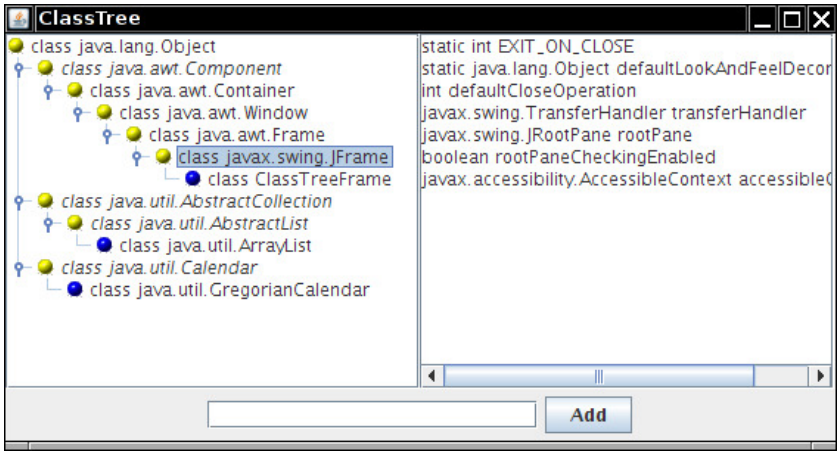


Figure 12.29: A class browser

To obtain this behavior, you need to install a tree selection listener. The listener must implement the TreeSelectionListener interface—an interface with a single method:

```
void valueChanged(TreeSelectionEvent event)
```

That method is called whenever the user selects or deselects tree nodes.

Add the listener to the tree in the normal way:

```
tree.addTreeSelectionListener(listener);
```

You can specify whether the user is allowed to select a single node, a contiguous range of nodes, or an arbitrary, potentially discontiguous, set of nodes. The JTree class uses a TreeSelectionModel to manage node selection. You need to retrieve the model to set the selection state to one of SINGLE_TREE_SELECTION, CONTIGUOUS_TREE_SELECTION, or DISCONTIGUOUS_TREE_SELECTION. (Discontiguous selection mode is the default.) For example, in our class browser, we want to allow selection of only a single class:

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

Apart from setting the selection mode, you need not worry about the tree selection model.

How the user selects multiple items depends on the look-and-feel. In the Metal look-and-feel, hold down the Ctrl key while clicking an item to add it to the selection, or to remove it if it is currently selected. Hold down the Shift key while clicking an item to select a range of items, extending from the previously selected item to the new item.

To find out the current selection, query the tree with the getSelectionPaths method:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

If you restricted the user to single-item selection, you can use the convenience method getSelectionPath which returns the first selected path or null if no path was selected.

The TreeSelectionEvent class has a getPaths method that returns an array of TreePath objects, but that array describes selection changes, not the current selection.

Listing 12.10 shows the frame class for the class tree program. The program displays inheritance hierarchies and customizes the display to show abstract classes in italics. (See Listing 12.11 for the cell renderer.) You can type the name of any class into the text field at the bottom of the frame. Press the Enter key or click the Add button to add the class and its superclasses to the tree. You must enter the full package name, such as java.util.ArrayList.

This program is a bit tricky because it uses reflection to construct the class tree. This work is done inside the addClass method. (The details are not that important. We use the class tree in this example because inheritance yields a nice supply of trees without laborious coding. When you display trees in your applications, you will have your own source of hierarchical

data.) The method uses the breadth-first search algorithm to find whether the current class is already in the tree by calling the findUserObject method that we implemented in the preceding section. If the class is not already in the tree, we add the superclasses to the tree, then make the new class node a child and make that node visible.

When you select a tree node, the text area to the right is filled with the fields of the selected class. In the frame constructor, we restrict the user to single-item selection and add a tree selection listener. When the valueChanged method is called, we ignore its event parameter and simply ask the tree for the current selection path. As always, we need to get the last node of the path and look up its user object. We then call the getFieldDescription method which uses reflection to assemble a string with all fields of the selected class.

Listing 12.10 **v2ch12/treeRender/ClassTreeFrame.java**

```
package v2ch12.treeRender;

import module java.base;
import module java.desktop;

/**
 * This frame displays the class tree, a text field, and an "Add" button to add mo
 * into the tree.
 */
public class ClassTreeFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;

   private DefaultMutableTreeNode root;
   private DefaultTreeModel model;
   private JTree tree;
   private JTextField textField;
   private JTextArea textArea;

   public ClassTreeFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
    // the root of the class tree is Object
    root = new DefaultMutableTreeNode(java.lang.Object.class);
    model = new DefaultTreeModel(root);
    tree = new JTree(model);

    // add this class to populate the tree with some data
    addClass(getClass());

    // set up node icons
    var renderer = new ClassNameTreeCellRenderer();
    renderer.setClosedIcon(new ImageIcon("images/red-ball.gif"));
    renderer.setOpenIcon(new ImageIcon("images/yellow-ball.gif"));
    renderer.setLeafIcon(new ImageIcon("images/blue-ball.gif"));
    tree.setCellRenderer(renderer);

    // set up selection mode
    tree.addTreeSelectionListener(_ -> {
       // the user selected a different node--update description
       TreePath path = tree.getSelectionPath();
       if (path == null) return;
       var selectedNode = (DefaultMutableTreeNode) path.getLastPathCompor
       Class<?> c = (Class<?>) selectedNode.getUserObject();
       String description = getFieldDescription(c);
       textArea.setText(description);
    });
    int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
    tree.getSelectionModel().setSelectionMode(mode);

    // this text area holds the class description
    textArea = new JTextArea();

    // add tree and text area
    var panel = new JPanel();
    panel.setLayout(new GridLayout(1, 2));
    panel.add(new JScrollPane(tree));
    panel.add(new JScrollPane(textArea));

    add(panel, BorderLayout.CENTER);

    addTextField();
}

/**
 * Add the text field and "Add" button to add a new class.
```

```
 */
public void addTextField() {
   var panel = new JPanel();

   ActionListener addListener = _ -> {
      // add the class whose name is in the text field
      try {
         String text = textField.getText();
         addClass(Class.forName(text)); // clear text field to indicate success
         textField.setText("");
      }
      catch (ClassNotFoundException e) {
         JOptionPane.showMessageDialog(null, "Class not found");
      }
   };

   // new class names are typed into this text field
   textField = new JTextField(20);
   textField.addActionListener(addListener);
   panel.add(textField);

   var addButton = new JButton("Add");
   addButton.addActionListener(addListener);
   panel.add(addButton);

   add(panel, BorderLayout.SOUTH);
}

/**
 * Finds an object in the tree.
 * @param obj the object to find
 * @return the node containing the object or null if the object is not present
 */
public DefaultMutableTreeNode findUserObject(Object obj) {
   // find the node containing a user object
   var e = (Enumeration<TreeNode>) root.breadthFirstEnumeration();
   while (e.hasMoreElements()) {
      var node = (DefaultMutableTreeNode) e.nextElement();
      if (node.getUserObject().equals(obj)) return node;
   }
   return null;
}

/**
```

```
 * Adds a new class and any parent classes that aren't yet part of the tree.
 * @param c the class to add
 * @return the newly added node
 */
public DefaultMutableTreeNode addClass(Class<?> c) {
   // add a new class to the tree

   // skip non-class types
   if (c.isInterface() || c.isPrimitive()) return null;

   // if the class is already in the tree, return its node
   DefaultMutableTreeNode node = findUserObject(c);
   if (node != null) return node;

   // class isn't present--first add class parent recursively

   Class<?> s = c.getSuperclass();

   DefaultMutableTreeNode parent;
   if (s == null)
      parent = root;
   else
      parent = addClass(s);

   // add the class as a child to the parent
   var newNode = new DefaultMutableTreeNode(c);
   model.insertNodeInto(newNode, parent, parent.getChildCount());

   // make node visible
   var path = new TreePath(model.getPathToRoot(newNode));
   tree.makeVisible(path);

   return newNode;
}

/**
 * Returns a description of the fields of a class.
 * @param c the class to be described
 * @return a string containing all field types and names
 */
public static String getFieldDescription(Class<?> c) {
   // use reflection to find types and names of fields
   var r = new StringBuilder();
   Field[] fields = c.getDeclaredFields();
```

```
      for (int i = 0; i < fields.length; i++) {
         Field f = fields[i];
         if (Modifier.isStatic(f.getModifiers())) r.append("static ");
         r.append(f.getType().getName());
         r.append(" ");
         r.append(f.getName());
         r.append("\n");
      }
      return r.toString();
   }
}
```

Listing 12.11 **v2ch12/treeRender/ClassNameTreeCellRenderer.java**

```
package v2ch12.treeRender;

import module java.base;
import module java.desktop;

/**
 * This class renders a class name either in plain or italic. Abstract classes are i
 */
public class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer {
   private Font plainFont = null;
   private Font italicFont = null;

   public Component getTreeCellRendererComponent(JTree tree, Object value,
         boolean expanded, boolean leaf, int row, boolean hasFocus) {
      super.getTreeCellRendererComponent(tree, value, selected, expanded, lea
            hasFocus);
      // get the user object
      var node = (DefaultMutableTreeNode) value;
      Class<?> c = (Class<?>) node.getUserObject();

      // the first time, derive italic font from plain font
      if (plainFont == null) {
         plainFont = getFont();
         // the tree cell renderer is sometimes called with a label that has a null f
         if (plainFont != null) italicFont = plainFont.deriveFont(Font.ITALIC);
      }

      // set font to italic if the class is abstract, plain otherwise
```

```
      if (Modifier.isAbstract(c.getModifiers()))
         setFont(italicFont);
      else
         setFont(plainFont);
      return this;
   }
}
```

javax.swing.JTree1.2

- TreePath getSelectionPath()
- TreePath[] getSelectionPaths()
  return the first selected path, or an array of paths to all selected nodes. If no paths are selected, both methods return null.

javax.swing.event.TreeSelectionListener1.2

- void valueChanged(TreeSelectionEvent event)
  is called whenever nodes are selected or deselected.

javax.swing.event.TreeSelectionEvent1.2

- TreePath getPath()
- TreePath[] getPaths()
  get the first path or all paths that have changed in this selection event. If you want to know the current selection, not the selection change, call JTree.getSelectionPaths instead.

12.4.6. Custom Tree Models

In the final example, we implement a program that inspects the contents of an object, just like a debugger does (see Figure 12.30).

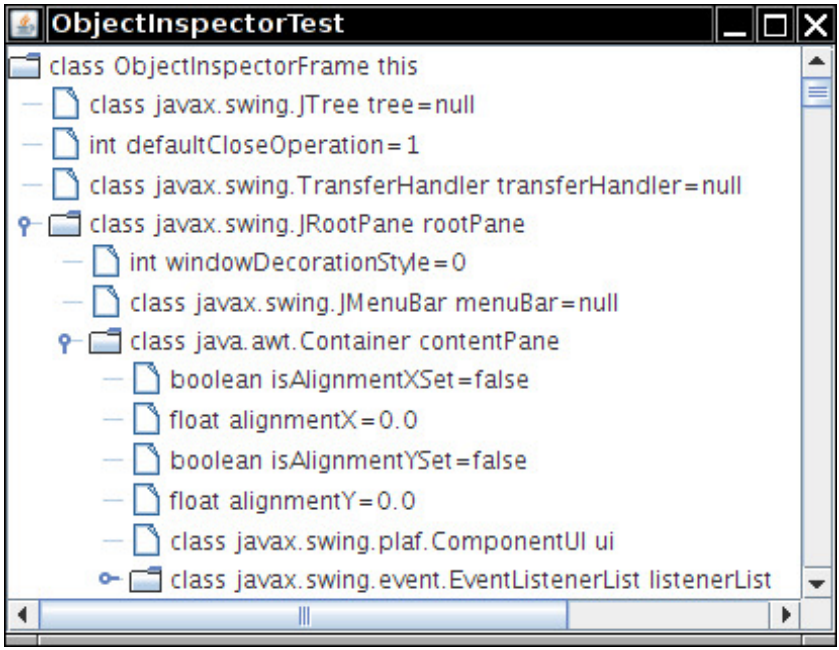


Figure 12.30: An object inspection tree

Before going further, compile and run the example program. Each node corresponds to an instance field. If the field is an object, expand it to see its instance fields. The program inspects the contents of the frame window. If you poke around a few of the instance fields, you should be able to find some familiar classes. You'll also gain some respect for how complex the Swing user interface components are under the hood.

What's remarkable about the program is that the tree does not use the DefaultTreeModel. If you already have data that are hierarchically organized, you might not want to build a duplicate tree and worry about keeping both trees synchronized. That is the situation in our case—the inspected objects are already linked to each other through the object references, so there is no need to replicate the linking structure.

The TreeModel interface has only a handful of methods. The first group of methods enables the JTree to find the tree nodes

by first getting the root, then the children. The JTree class calls these methods only when the user actually expands a node.

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

This example shows why the TreeModel interface, like the JTree class itself, does not need an explicit notion of nodes. The root and its children can be any objects. The TreeModel is responsible for telling the JTree how they are connected.

The next method of the TreeModel interface is the reverse of getChild:

```
int getIndexOfChild(Object parent, Object child)
```

Actually, this method can be implemented in terms of the first three—see the code in Listing 12.12.

The tree model tells the JTree which nodes should be displayed as leaves:

```
boolean isLeaf(Object node)
```

If your code changes the tree model, the tree needs to be notified so it can redraw itself. The tree adds itself as a TreeModelListener to the model. Thus, the model must support the usual listener management methods:

```
void addTreeModelListener(TreeModelListener l)
void removeTreeModelListener(TreeModelListener l)
```

You can see the implementations for these methods in Listing 12.13.

When the model modifies the tree contents, it calls one of the four methods of the TreeModelListener interface:

```
void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodesRemoved(TreeModelEvent e)
```

```
void treeStructureChanged(TreeModelEvent e)
```

The TreeModelEvent object describes the location of the change. The details of assembling a tree model event that describes an insertion or removal event are quite technical. You only need to worry about firing these events if your tree can actually have nodes added and removed. In Listing 12.12, we show how to fire one event by replacing the root with a new object.

To simplify the code for event firing, use the javax.swing.EventListenerList convenience class that collects listeners. The last three methods of Listing 12.13 show how to use the class.

Finally, if the user edits a tree node, your model is called with the change:

```
void valueForPathChanged(TreePath path, Object newValue)
```

If you don't allow editing, this method is never called.

If you don't need to support editing, constructing a tree model is easily done. Implement the three methods:

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

These methods describe the structure of the tree. Supply routine implementations of the other five methods, as in Listing 12.12. You are then ready to display your tree.

Now let's turn to the implementation of the example program. Our tree will contain objects of type Variable.

Had we used the DefaultTreeModel, our nodes would have been objects of type DefaultMutableTreeNode with user objects of type Variable.

For example, suppose you inspect the variable

```
Employee joe;
```

That variable has a type Employee.class, a name "joe", and a value—the value of the object reference joe. In Listing 12.14, we define a class Variable that describes a variable in a program:

```
var v = new Variable(Employee.class, "joe", joe);
```

If the type of the variable is a primitive type, you must use an object wrapper for the value.

```
new Variable(double.class, "salary", Double.valueOf(salary));
```

If the type of the variable is a class, the variable has fields. Using reflection, we enumerate all fields and collect them in an ArrayList. Since the getFields method of the Class class does not return the fields of the superclass, we need to call getFields on all superclasses as well. You can find the code in the Variable constructor. The getFields method of our Variable class returns the array of fields. Finally, the toString method of the Variable class formats the node label. The label always contains the variable type and name. If the variable is not a class, the label also contains the value.

> If the type is an array, we do not display the elements of the array. This would not be difficult to do; I leave it as the proverbial “exercise for the reader.”

Let’s move on to the tree model. The first two methods are simple.

```
public Object getRoot() {
   return root;
}

public int getChildCount(Object parent) {
   return ((Variable) parent).getFields().size();
}
```

The getChild method returns a new Variable object that describes the field with the given index. The getType and getName methods of the Field class yield the field type and name. By using reflection, you can read the field value as f.get(parentValue). That method can throw an IllegalAccessException. However, we made all fields accessible in the Variable constructor, so this won't happen in practice.

Here is the complete code of the getChild method:

```
public Object getChild(Object parent, int index) {
   ArrayList fields = ((Variable) parent).getFields();
   var f = (Field) fields.get(index);
   Object parentValue = ((Variable) parent).getValue();
   try {
      return new Variable(f.getType(), f.getName(), f.get(parentValue)
   }
   catch (IllegalAccessException e) {
      return null;
   }
}
```

These three methods reveal the structure of the object tree to the JTree component. The remaining methods are routine—see the source code in Listing 12.13.

There is one remarkable fact about this tree model: It actually describes an infinite tree. You can verify this by following one of the WeakReference objects. Click on the variable named referent. It leads you right back to the original object. You get an identical subtree, and you can open its WeakReference object again, ad infinitum. Of course, you cannot store an infinite set of nodes; the tree model simply generates the nodes on demand as the user expands the parents. Listing 12.12 shows the frame class of the sample program.

Listing 12.12 **v2ch12/treeModel/ObjectInspectorFrame.java**

```
package v2ch12.treeModel;
```

```
import module java.desktop;

/**
 * This frame holds the object tree.
 */
public class ObjectInspectorFrame extends JFrame {
   private JTree tree;
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 300;

   public ObjectInspectorFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // we inspect this frame object

      var v = new Variable(getClass(), "this", this);
      var model = new ObjectTreeModel();
      model.setRoot(v);

      // construct and show tree

      tree = new JTree(model);
      add(new JScrollPane(tree), BorderLayout.CENTER);
   }
}
```

Listing 12.13 **v2ch12/treeModel/ObjectTreeModel.java**

```
package v2ch12.treeModel;

import module java.base;
import module java.desktop;

/**
 * This tree model describes the tree structure of a Java object. Children are th
 * are stored in instance variables.
 */
public class ObjectTreeModel implements TreeModel {
   private Variable root;
   private EventListenerList listenerList = new EventListenerList();

   /**
```

```
 * Constructs an empty tree.
 */
public ObjectTreeModel() {
   root = null;
}

/**
 * Sets the root to a given variable.
 * @param v the variable that is being described by this tree
 */
public void setRoot(Variable v) {
   Variable oldRoot = v;
   root = v;
   fireTreeStructureChanged(oldRoot);
}

public Object getRoot() {
   return root;
}

public int getChildCount(Object parent) {
   return ((Variable) parent).getFields().size();
}

public Object getChild(Object parent, int index) {
   ArrayList<Field> fields = ((Variable) parent).getFields();
   var f = (Field) fields.get(index);
   Object parentValue = ((Variable) parent).getValue();
   try {
      return new Variable(f.getType(), f.getName(), f.get(parentValue));
   }
   catch (IllegalAccessException e) {
      return null;
   }
}

public int getIndexOfChild(Object parent, Object child) {
   int n = getChildCount(parent);
   for (int i = 0; i < n; i++)
      if (getChild(parent, i).equals(child)) return i;
   return -1;
}

public boolean isLeaf(Object node) {
```

```
      return getChildCount(node) == 0;
   }

   public void valueForPathChanged(TreePath path, Object newValue) {
   }

   public void addTreeModelListener(TreeModelListener l) {
      listenerList.add(TreeModelListener.class, l);
   }

   public void removeTreeModelListener(TreeModelListener l) {
      listenerList.remove(TreeModelListener.class, l);
   }

   protected void fireTreeStructureChanged(Object oldRoot) {
      var event = new TreeModelEvent(this, new Object[] { oldRoot });
      for (TreeModelListener l : listenerList.getListeners(TreeModelListener.clas
         l.treeStructureChanged(event);
   }
}
```

Listing 12.14 **v2ch12/treeModel/Variable.java**

```
package v2ch12.treeModel;

import module java.base;

/**
 * A variable with a type, name, and value.
 */
public class Variable {
   private Class<?> type;
   private String name;
   private Object value;
   private ArrayList<Field> fields;

   /**
    * Construct a variable.
    * @param aType the type
    * @param aName the name
    * @param aValue the value
    */
   public Variable(Class<?> aType, String aName, Object aValue) {
```

```
      type = aType;
      name = aName;
      value = aValue;
      fields = new ArrayList<>();

      // find all fields if we have a class type except we don't expand strings and
      // null values

      if (!type.isPrimitive() && !type.isArray() && !type.equals(String.class)
            && value != null) {
         // get fields from the class and all superclasses
         for (Class<?> c = value.getClass(); c != null; c = c.getSuperclass()) {
            Field[] fs = c.getDeclaredFields();
            AccessibleObject.setAccessible(fs, true);

            // get all nonstatic fields
            for (Field f : fs)
               if (!Modifier.isStatic(f.getModifiers())) fields.add(f);
         }
      }
   }

   /**
    * Gets the value of this variable.
    * @return the value
    */
   public Object getValue() {
      return value;
   }

   /**
    * Gets all nonstatic fields of this variable.
    * @return an array list of variables describing the fields
    */
   public ArrayList<Field> getFields() {
      return fields;
   }

   public String toString() {
      String r = type + " " + name;
      if (type.isPrimitive())
         r += "=" + value;
      else if (type.equals(String.class))
         r += "=" + value;
```

```
      else if (value == null) r += "=null";
      return r;
   }
}
```

javax.swing.tree.TreeModel 1.2

- Object getRoot()
  returns the root node.
- int getChildCount(Object parent)
  gets the number of children of the parent node.
- Object getChild(Object parent, int index)
  gets the child node of the parent node at the given index.
- int getIndexOfChild(Object parent, Object child)
  gets the index of the child node in the parent node, or -1 if child is not a child of parent in this tree model.
- boolean isLeaf(Object node)
  returns true if node is conceptually a leaf of the tree.
- void addTreeModelListener(TreeModelListener l)
- void removeTreeModelListener(TreeModelListener l)
  add or remove listeners that are notified when the information in the tree model changes.
- void valueForPathChanged(TreePath path, Object newValue)
  is called when a cell editor has modified the value of a node.

javax.swing.event.TreeModelListener 1.2

- void treeNodesChanged(TreeModelEvent e)
- void treeNodesInserted(TreeModelEvent e)
- void treeNodesRemoved(TreeModelEvent e)
- void treeStructureChanged(TreeModelEvent e)
  are called by the tree model when the tree has been modified.

javax.swing.event.TreeModelEvent 1.2

- TreeModelEvent(Object eventSource, TreePath node) constructs a tree model event.

12.5. Advanced AWT

You can use the methods of the Graphics class to create simple drawings. Those methods are sufficient for simple applications, but they fall short when you need to create complex shapes or require complete control over the appearance of the graphics. The Java 2D API is a more sophisticated class library that you can use to produce high-quality drawings. In the following sections, I'll give you an overview of that API.

12.5.1. The Rendering Pipeline

The original JDK 1.0 had a very simple mechanism for drawing shapes. You selected color and paint mode, and called methods of the Graphics class such as drawRect or fillOval. The Java 2D API supports many more options.

- You can easily produce a wide variety of shapes.
- You have control over the stroke—the pen that traces shape boundaries.
- You can fill shapes with solid colors, varying hues, and repeating patterns.
- You can use transformations to move, scale, rotate, or stretch shapes.
- You can clip shapes to restrict them to arbitrary areas.
- You can select composition rules to describe how to combine the pixels of a new shape with existing pixels.

To draw a shape, go through the following steps:

1. Obtain an object of the Graphics2D class. This class is a subclass of the Graphics class. Ever since Java 1.2, methods such as paint and paintComponent automatically receive an object of the Graphics2D class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g) {
   var g2 = (Graphics2D) g;
   . . .
}
```

2. Use the setRenderingHints method to set rendering hints—trade-offs between speed and drawing quality.

```
RenderingHints hints = . . .;
g2.setRenderingHints(hints);
```

3. Use the setStroke method to set the stroke. The stroke draws the outline of the shape. You can select the thickness and choose among solid and dotted lines.

```
Stroke stroke = . . .;
g2.setStroke(stroke);
```

4. Use the setPaint method to set the paint. The paint fills areas such as the stroke or the interior of a shape. You can create solid color paint, paint with changing hues, or tiled fill patterns.

```
Paint paint = . . .;
g2.setPaint(paint);
```

5. Use the clip method to set the clipping region.

```
Shape clip = . . .;
g2.clip(clip);
```

6. Use the transform method to set a transformation from user space to device space. Use transformations if it is easier for you to define your shapes in a custom coordinate system than by using pixel coordinates.

```
AffineTransform transform = . . .;
g2.transform(transform);
```

7. Use the setComposite method to set a composition rule that describes how to combine the new pixels with the existing pixels.

   ```
   Composite composite = . . .;
   g2.setComposite(composite);
   ```

8. Create a shape. The Java 2D API supplies many shape objects and methods to combine shapes.

   ```
   Shape shape = . . .;
   ```

9. Draw or fill the shape. If you draw the shape, its outline is stroked. If you fill the shape, the interior is painted.

   ```
   g2.draw(shape);
   g2.fill(shape);
   ```

Of course, in many practical circumstances, you don't need all these steps. There are reasonable defaults for the settings of the 2D graphics context; change the settings only if you want to deviate from the defaults.

In the following sections, you will see how to describe shapes, strokes, paints, transformations, and composition rules.

The various set methods simply set the state of the 2D graphics context. They don't cause any drawing. Similarly, when you construct Shape objects, no drawing takes place. A shape is only rendered when you call draw or fill. At that time, the new shape is computed in a rendering pipeline (see Figure 12.31).

.

Figure 12.31: The rendering pipeline

In the rendering pipeline, the following steps take place to render a shape:

1. The path of the shape is stroked.

2. The shape is transformed.
3. The shape is clipped. If there is no intersection between the shape and the clipping area, the process stops.
4. The remainder of the shape after clipping is filled.
5. The pixels of the filled shape are composed with the existing pixels. (In Figure 12.31, the circle is part of the existing pixels, and the cup shape is superimposed over it.)

In the next section, you will see how to define shapes. Then, we will turn to the 2D graphics context settings.

java.awt.Graphics2D 1.2

- void draw(Shape s)
  draws the outline of the given shape with the current paint.
- void fill(Shape s)
  fills the interior of the given shape with the current paint.

12.5.2. The Shape Class Hierarchy

Here are some of the methods in the Graphics class to draw shapes:

```
drawLine
drawRectangle
drawRoundRect
draw3DRect
drawPolygon
drawPolyline
drawOval
drawArc
```

There are also corresponding fill methods. These methods have been in the Graphics class ever since JDK 1.0. The Java 2D API

uses a completely different, object-oriented approach. Instead of methods, there are classes:

Line2D
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
QuadCurve2D
CubicCurve2D
GeneralPath

These classes all implement the Shape interface, which we will examine in the following sections.

The Line2D, Rectangle2D, RoundRectangle2D, Ellipse2D, and Arc2D classes correspond to the drawLine, drawRectangle, drawRoundRect, drawOval, and drawArc methods. (The concept of a “3D rectangle” has died the death it so richly deserved—there is no analog to the draw3DRect method.) The Java 2D API supplies two additional classes, quadratic and cubic curves, that we will discuss in this section. There is no Polygon2D class; instead, the GeneralPath class describes paths made up from lines, quadratic and cubic curves. You can use a GeneralPath to describe a polygon; we'll show you how later in this section.

To draw a shape, first create an object of a class that implements the Shape interface and then call the draw method of the Graphics2D class.

The classes

Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D

all inherit from a common superclass RectangularShape. Admittedly, ellipses and arcs are not rectangular, but they have a bounding rectangle (see Figure 12.32).

.

Figure 12.32: The bounding rectangle of an ellipse and an arc

Each of the classes with a name ending in “2D” has two subclasses for specifying coordinates as float or double quantities. In Volume I, you already encountered Rectangle2D.Float and Rectangle2D.Double.

The same scheme is used for the other classes, such as Arc2D.Float and Arc2D.Double.

Internally, all graphics classes use float coordinates because float numbers use less storage space but have sufficient precision for geometric computations. However, the Java programming language makes it a bit more tedious to manipulate float numbers. For that reason, most methods of the graphics classes use double parameters and return values. Only when constructing a 2D object you need to choose between the constructors with float and double coordinates. For example,

```
var floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
var doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

The Xxx2D.Float and Xxx2D.Double classes are subclasses of the Xxx2D classes. After object construction, essentially no benefit accrues from remembering the subclass, and you can just store the constructed object in a superclass variable as in the code example above.

As you can see from the curious names, the Xxx2D.Float and Xxx2D.Double classes are also inner classes of the Xxx2D classes. That is just a minor syntactical convenience to avoid inflation of outer class names.

Finally, the Point2D class describes a point with an x and a y coordinate. Points are used to define shapes, but they aren’t themselves shapes.

Figure 12.33 shows the relationships between the shape classes. However, the Double and Float subclasses are omitted. Legacy classes from the pre-2D library are marked with a gray fill.

.

Figure 12.33: Relationships between the shape classes

12.5.3. Constructing Shapes

You already saw how to use the Rectangle2D, Ellipse2D, and Line2D classes in Chapter 10. In this section, you will learn how to work with the remaining 2D shapes.

For the RoundRectangle2D shape, specify the top left corner, width, height, and the x and y dimensions of the corner area that should be rounded (see Figure 12.34). For example, the call

```
var r = new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20);
```

produces a rounded rectangle with circles of radius 20 at each of the corners.

.

Figure 12.34: Constructing a RoundRectangle2D

To construct an arc, specify the bounding box, the start angle, the angle swept out by the arc (see Figure 12.35), and the closure type—one of Arc2D.OPEN, Arc2D.PIE, or Arc2D.CHORD.

```
var a = new Arc2D(x, y, width, height, startAngle, arcAngle, closureT
```

.

Figure 12.35: Constructing an elliptical arc

Figure 12.36 illustrates the arc types.

.

Figure 12.36: Arc types

If the arc is elliptical, the computation of the arc angles is not at all straightforward. The API documentation states: "The angles are specified relative to the nonsquare framing rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the framing rectangle. As a result, if the framing rectangle is noticeably longer along one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the frame." Unfortunately, the documentation is silent on how to compute this "skew." Here are the details:

Suppose the center of the arc is the origin and the point (x, y) lies on the arc. You can get a skewed angle with the following formula:

```
skewedAngle = Math.toDegrees(Math.atan2(-y * height, x * wi
```

The result is a value between -180 and 180. Compute the skewed start and end angles in this way. Then, compute the difference between the two skewed angles. If the start angle or the difference is negative, add 360 to the start angle. Then, supply the start angle and the difference to the arc constructor.

If you run the example program at the end of this section, you can visually check that this calculation yields the correct values for the arc constructor (see Figure 12.39).

The Java 2D API supports quadratic and cubic curves. In this chapter, we do not get into the mathematics of these curves. You can get a feel for how the curves look by running the program in Listing 12.15. Quadratic curves are specified by two end points and a control point (see Figure 12.37). Cubic curves (Figure 12.38) have two control points. Moving the control

points changes the shape of the curves.

.

Figure 12.37: A quadratic curve

.

Figure 12.38: A cubic curve

To construct quadratic and cubic curves, give the coordinates of the end points and the control points. For example,

```
var q = new QuadCurve2D.Double(startX, startY, controlX, controlY, e
var c = new CubicCurve2D.Double(startX, startY, control1X, control1
   control2X, control2Y, endX, endY);
```

Quadratic curves are not very flexible, and they are not commonly used in practice. Cubic curves (such as the Bézier curves drawn by the CubicCurve2D class) are, however, very common. By combining many cubic curves so that the slopes at the connection points match, you can create complex, smooth-looking curved shapes. For more information, refer to Computer Graphics: Principles and Practice, Third Edition, by John F. Hughes et al.

You can build arbitrary sequences of line segments, quadratic curves, and cubic curves, and store them in a GeneralPath object. Specify the first coordinate of the path with the moveTo method, for example:

```
var path = new GeneralPath();
path.moveTo(10, 20);
```

You can then extend the path by calling one of the methods lineTo, quadTo, or curveTo. These methods extend the path by a line, a quadratic curve, or a cubic curve. To call lineTo, supply the end point. For the two curve methods, supply the control points, then the end point. For example,

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y, endX, endY
```

Close the path by calling the closePath method. It draws a straight line back to the starting point of the path.

To make a polygon, simply call moveTo to go to the first corner point, followed by repeated calls to lineTo to visit the other corner points. Finally, call closePath to close the polygon. The program in Listing 12.15 shows this in more detail.

A general path does not have to be connected. You can call moveTo at any time to start a new path segment.

Finally, you can use the append method to add arbitrary Shape objects to a general path. The outline of the shape is added to the end to the path. The second parameter of the append method is true if the new shape should be connected to the last point on the path, false otherwise. For example, the call

```
Rectangle2D r = . . .;
path.append(r, false);
```

appends the outline of a rectangle to the path without connecting it to the existing path. But

```
path.append(r, true);
```

adds a straight line from the end point of the path to the starting point of the rectangle, and then adds the rectangle outline to the path.

The program in Listing 12.15 lets you create sample paths. Pick a shape maker from the combo box. The program contains shape makers for

- Straight lines
- Rectangles, rounded rectangles, and ellipses
- Arcs (showing lines for the bounding rectangle and the start and end angles, in addition to the arc itself)
- Polygons (using a GeneralPath)
- Quadratic and cubic curves

Use the mouse to adjust the control points. As you move them, the shape continuously repaints itself.

The program is a bit complex because it handles multiple shapes and supports dragging of the control points.

An abstract superclass ShapeMaker encapsulates the commonality of the shape maker classes. Each shape has a fixed number of control points that the user can move around. The getPointCount method returns that value. The abstract method

```
Shape makeShape(Point2D[] points)
```

computes the actual shape, given the current positions of the control points. The toString method returns the class name so that the ShapeMaker objects can simply be dumped into a JComboBox.

To enable dragging of the control points, the ShapePanel class handles both mouse and mouse motion events. If the mouse is pressed on top of a rectangle, subsequent mouse drags move the rectangle.

The majority of the shape maker classes are simple—their makeShape methods just construct and return the requested shapes. However, the ArcMaker class needs to compute the distorted start and end angles. Furthermore, to demonstrate that the computation is indeed correct, the returned shape is a GeneralPath containing the arc itself, the bounding rectangle, and the lines from the center of the arc to the angle control points (see Figure 12.39).

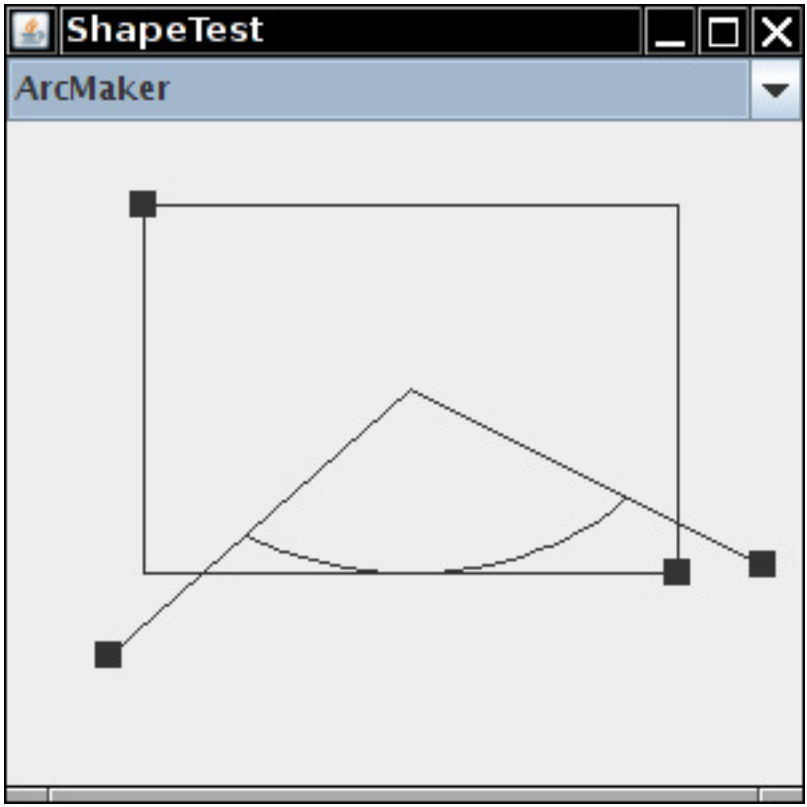


Figure 12.39: The ShapeDemo program

Listing 12.15 **v2ch12/shape/ShapeDemo.java**

```
package v2ch12.shape;

import module java.base;
import module java.desktop;
import java.awt.event.MouseEvent;

/**
 * This program demonstrates the various 2D shapes.
 */
class ShapeDemo {
   void main() {
      EventQueue.invokeLater(() -> {
```

```
            var frame = new ShapeTestFrame();
            frame.setTitle("ShapeTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
         });
      }
   }

   /**
    * This frame contains a combo box to select a shape and a component to draw
    */
   class ShapeTestFrame extends JFrame {
      public ShapeTestFrame() {
         var comp = new ShapeComponent();
         add(comp, BorderLayout.CENTER);
         var comboBox = new JComboBox<ShapeMaker>();
         comboBox.addItem(new LineMaker());
         comboBox.addItem(new RectangleMaker());
         comboBox.addItem(new RoundRectangleMaker());
         comboBox.addItem(new EllipseMaker());
         comboBox.addItem(new ArcMaker());
         comboBox.addItem(new PolygonMaker());
         comboBox.addItem(new QuadCurveMaker());
         comboBox.addItem(new CubicCurveMaker());
         comboBox.addActionListener(_ -> {
            ShapeMaker shapeMaker = comboBox.getItemAt(comboBox.getSelected
            comp.setShapeMaker(shapeMaker);
         });
         add(comboBox, BorderLayout.NORTH);
         comp.setShapeMaker((ShapeMaker) comboBox.getItemAt(0));
         pack();
      }
   }

   /**
    * This component draws a shape and allows the user to move the points that d
    */
   class ShapeComponent extends JComponent {
      private static final Dimension PREFERRED_SIZE = new Dimension(300, 200);
      private Point2D[] points;
      private static RandomGenerator generator = RandomGenerator.getDefault();
      private static int SIZE = 10;
      private int current;
      private ShapeMaker shapeMaker;
```

```
public ShapeComponent() {
   addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent event) {
         Point p = event.getPoint();
         for (int i = 0; i < points.length; i++) {
            double x = points[i].getX() - SIZE / 2;
            double y = points[i].getY() - SIZE / 2;
            var r = new Rectangle2D.Double(x, y, SIZE, SIZE);
            if (r.contains(p)) {
               current = i;
               return;
            }
         }
      }

      public void mouseReleased(MouseEvent event) {
         current = -1;
      }
   });
   addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent event) {
         if (current == -1) return;
         points[current] = event.getPoint();
         repaint();
      }
   });
   current = -1;
}

/**
 * Set a shape maker and initialize it with a random point set.
 * @param aShapeMaker a shape maker that defines a shape from a point se
 */
public void setShapeMaker(ShapeMaker aShapeMaker) {
   shapeMaker = aShapeMaker;
   int n = shapeMaker.getPointCount();
   points = new Point2D[n];
   for (int i = 0; i < n; i++) {
      double x = generator.nextDouble() * getWidth();
      double y = generator.nextDouble() * getHeight();
      points[i] = new Point2D.Double(x, y);
   }
   repaint();
```

```
    }

    public void paintComponent(Graphics g) {
        if (points == null) return;
        var g2 = (Graphics2D) g;
        for (int i = 0; i < points.length; i++) {
            double x = points[i].getX() - SIZE / 2;
            double y = points[i].getY() - SIZE / 2;
            g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
        }

        g2.draw(shapeMaker.makeShape(points));
    }

    public Dimension getPreferredSize() {
        return PREFERRED_SIZE;
    }
}

/**
 * A shape maker can make a shape from a point set. Concrete subclasses must
 * the makeShape method.
 */
abstract class ShapeMaker {
    private int pointCount;

    /**
     * Constructs a shape maker.
     * @param pointCount the number of points needed to define this shape
     */
    public ShapeMaker(int pointCount) {
        this.pointCount = pointCount;
    }

    /**
     * Gets the number of points needed to define this shape.
     * @return the point count
     */
    public int getPointCount() {
        return pointCount;
    }

    /**
     * Makes a shape out of the given point set.
```

```
     * @param p the points that define the shape
     * @return the shape defined by the points
     */
    public abstract Shape makeShape(Point2D[] p);

    public String toString() {
       return getClass().getName();
    }
}

/**
 * Makes a line that joins two given points.
 */
class LineMaker extends ShapeMaker {
    public LineMaker() {
       super(2);
    }

    public Shape makeShape(Point2D[] p) {
       return new Line2D.Double(p[0], p[1]);
    }
}

/**
 * Makes a rectangle that joins two given corner points.
 */
class RectangleMaker extends ShapeMaker {
    public RectangleMaker() {
       super(2);
    }

    public Shape makeShape(Point2D[] p) {
       var s = new Rectangle2D.Double();
       s.setFrameFromDiagonal(p[0], p[1]);
       return s;
    }
}

/**
 * Makes a round rectangle that joins two given corner points.
 */
class RoundRectangleMaker extends ShapeMaker {
    public RoundRectangleMaker() {
       super(2);
```

```
   }

   public Shape makeShape(Point2D[] p) {
      var s = new RoundRectangle2D.Double(0, 0, 0, 0, 20, 20);
      s.setFrameFromDiagonal(p[0], p[1]);
      return s;
   }
}

/**
 * Makes an ellipse contained in a bounding box with two given corner points.
 */
class EllipseMaker extends ShapeMaker {
   public EllipseMaker() {
      super(2);
   }

   public Shape makeShape(Point2D[] p) {
      var s = new Ellipse2D.Double();
      s.setFrameFromDiagonal(p[0], p[1]);
      return s;
   }
}

/**
 * Makes an arc contained in a bounding box with two given corner points, an
 * ending angles given by lines emanating from the center of the bounding box
 * given points. To show the correctness of the angle computation, the returned
 * the arc, the bounding box, and the lines.
 */
class ArcMaker extends ShapeMaker {
   public ArcMaker() {
      super(4);
   }

   public Shape makeShape(Point2D[] p) {
      double centerX = (p[0].getX() + p[1].getX()) / 2;
      double centerY = (p[0].getY() + p[1].getY()) / 2;
      double width = Math.abs(p[1].getX() - p[0].getX());
      double height = Math.abs(p[1].getY() - p[0].getY());

      double skewedStartAngle = Math.toDegrees(
         Math.atan2(-(p[2].getY() - centerY) * width, (p[2].getX() - centerX) * heigh
      double skewedEndAngle = Math.toDegrees(
```

```
         Math.atan2(-(p[3].getY() - centerY) * width, (p[3].getX() - centerX) * heigh
      double skewedAngleDifference = skewedEndAngle - skewedStartAngle;
      if (skewedStartAngle < 0) skewedStartAngle += 360;
      if (skewedAngleDifference < 0) skewedAngleDifference += 360;

      var s = new Arc2D.Double(0, 0, 0, 0, skewedStartAngle, skewedAngleDiffer
            Arc2D.OPEN);
      s.setFrameFromDiagonal(p[0], p[1]);

      var g = new GeneralPath();
      g.append(s, false);
      var r = new Rectangle2D.Double();
      r.setFrameFromDiagonal(p[0], p[1]);
      g.append(r, false);
      var center = new Point2D.Double(centerX, centerY);
      g.append(new Line2D.Double(center, p[2]), false);
      g.append(new Line2D.Double(center, p[3]), false);
      return g;
   }
}

/**
 * Makes a polygon defined by six corner points.
 */
class PolygonMaker extends ShapeMaker {
   public PolygonMaker() {
      super(6);
   }

   public Shape makeShape(Point2D[] p) {
      var s = new GeneralPath();
      s.moveTo((float) p[0].getX(), (float) p[0].getY());
      for (int i = 1; i < p.length; i++)
         s.lineTo((float) p[i].getX(), (float) p[i].getY());
      s.closePath();
      return s;
   }
}

/**
 * Makes a quad curve defined by two end points and a control point.
 */
class QuadCurveMaker extends ShapeMaker {
   public QuadCurveMaker() {
```

```
      super(3);
   }

   public Shape makeShape(Point2D[] p) {
      return new QuadCurve2D.Double(p[0].getX(), p[0].getY(), p[1].getX(), p[1].g
            p[2].getX(), p[2].getY());
   }
}

/**
 * Makes a cubic curve defined by two end points and two control points.
 */
class CubicCurveMaker extends ShapeMaker {
   public CubicCurveMaker() {
      super(4);
   }

   public Shape makeShape(Point2D[] p) {
      return new CubicCurve2D.Double(p[0].getX(), p[0].getY(), p[1].getX(), p[1].g
            p[2].getX(), p[2].getY(), p[3].getX(), p[3].getY());
   }
}
```

java.awt.geom.RoundRectangle2D.Double1.2

- RoundRectangle2D.Double(double x, double y, double width, double height, double arcWidth, double arcHeight) constructs a rounded rectangle with the given bounding rectangle and arc dimensions. See for an explanation of the arcWidth and arcHeight parameters.

java.awt.geom.Arc2D.Double1.2

- Arc2D.Double(double x, double y, double w, double h, double startAngle, double arcAngle, int type) constructs an arc with the given bounding rectangle, start and arc angle, and arc type. The startAngle and arcAngle are shown in . The type is one of Arc2D.OPEN, Arc2D.PIE, and Arc2D.CHORD.

java.awt.geom.QuadCurve2D.Double1.2

- QuadCurve2D.Double(double x1, double y1, double ctrlx, double ctrly, double x2, double y2)
  constructs a quadratic curve from a start point, a control point, and an end point.

java.awt.geom.CubicCurve2D.Double1.2

- CubicCurve2D.Double(double x1, double y1, double ctrlx1, double ctrly1, double ctrlx2, double ctrly2, double x2, double y2)
  constructs a cubic curve from a start point, two control points, and an end point.

java.awt.geom.GeneralPath1.2

- GeneralPath()
  constructs an empty general path.

java.awt.geom.Path2D.Float6

- void moveTo(float x, float y)
  makes (x, y) the current point—that is, the starting point of the next segment.
- void lineTo(float x, float y)
- void quadTo(float ctrlx, float ctrly, float x, float y)
- void curveTo(float ctrl1x, float ctrl1y, float ctrl2x, float ctrl2y, float x, float y)
  draw a line, quadratic curve, or cubic curve from the current point to the end point (x, y), and make that end point the current point.

java.awt.geom.Path2D6

- void append(Shape s, boolean connect)
  adds the outline of the given shape to the general path. If connect is true, the current point of the general path is connected to the starting point of the added shape by a straight line.
- void closePath()
  closes the path by drawing a straight line from the current point to the first point in the path.

12.5.4. Areas

In the preceding section, you saw how you can specify complex shapes by constructing general paths composed of lines and curves. By using a sufficient number of lines and curves, you can draw essentially any shape. For example, the shapes of characters in the fonts that you see on the screen and on your printouts are made up of lines and quadratic or cubic curves.

Occasionally, it is easier to describe a shape by composing it from areas, such as rectangles, polygons, or ellipses. The Java 2D API supports four constructive area geometry operations that combine two areas into a new area.

- add: The combined area contains all points that are in the first or the second area.
- subtract: The combined area contains all points that are in the first but not the second area.
- intersect: The combined area contains all points that are in the first and the second area.
- exclusiveOr: The combined area contains all points that are in either the first or the second area, but not in both.

Figure 12.40 shows these operations.

.

Figure 12.40: Constructive area geometry operations

To construct a complex area, start with a default area object.

```
var a = new Area();
```

Then, combine the area with any shape.

```
a.add(new Rectangle2D.Double(. . .));
a.subtract(path);
. . .
```

The Area class implements the Shape interface. You can stroke the boundary of the area with the draw method or paint the interior with the fill method of the Graphics2D class.

java.awt.geom.Area
■ void add(Area other) ■ void subtract(Area other) ■ void intersect(Area other) ■ void exclusiveOr(Area other) carry out the constructive area geometry operation with this area and the other area and set this area to the result.

12.5.5. Strokes

The draw operation of the Graphics2D class draws the boundary of a shape by using the currently selected stroke. By default, the stroke is a solid line that is 1 pixel wide. You can select a different stroke by calling the setStroke method and supplying an object of a class that implements the Stroke interface. The Java 2D API defines only one such class, called BasicStroke. In this section, we'll look at the capabilities of the BasicStroke class.

You can construct strokes of arbitrary thickness. For example, here is how to draw lines that are 10 pixels wide:

```
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Line2D.Double(. . .));
```

When a stroke is more than a pixel thick, the end of the stroke can have different styles. Figure 12.41 shows these so-called end cap styles. You have three choices:

- A butt cap simply ends the stroke at its end point.
- A round cap adds a half-circle to the end of the stroke.
- A square cap adds a half-square to the end of the stroke.

.

Figure 12.41: End cap styles

When two thick strokes meet, there are three choices for the join style (see Figure 12.42).

- A bevel join joins the strokes with a straight line that is perpendicular to the bisector of the angle between the two strokes.
- A round join extends each stroke to have a round cap.
- A miter join extends both strokes by adding a "spike."

.

Figure 12.42: Join styles

If two lines come together in a miter join at a very small angle, a bevel join is used instead, preventing extremely long spikes. The miter limit controls this transition. Technically, this is the ratio of the distance of the inner and outer corners of the spike divided by the stroke width. The default miter limit of 10 corresponds to an angle of about 11 degrees.

You can specify these choices in the BasicStroke constructor, for example:

```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND, BasicS
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, BasicStro
   15.0F /* miter limit */));
```

Finally, you can create dashed lines by setting a dash pattern. In the program in Listing 12.16, you can select a dash pattern that spells out SOS in Morse code. The dash pattern is a float[] array that contains the lengths of the "on" and "off" intervals (see Figure 12.43).

.

Figure 12.43: A dash pattern

You can specify the dash pattern and a dash phase when constructing the BasicStroke. The dash phase indicates where in the dash pattern each line should start. A segment of this length, preceding the starting point of the stroke, is assumed to have the dash pattern already applied. Normally, you set this value to 0.

```
float[] dashPattern = { 10, 10, 10, 10, 10, 10, 30, 10, 30, . . . };
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT, BasicStro
   10.0F /* miter limit */, dashPattern, 0 /* dash phase */));
```

End cap styles are applied to the ends of each dash in a dash pattern.

The program in Listing 12.16 lets you specify end cap styles, join styles, and dashed lines (see Figure 12.44). You can move the ends of the line segments to test the miter limit: Select the miter join, then move the line segment to form a very acute angle. You will see the miter join turn into a bevel join.

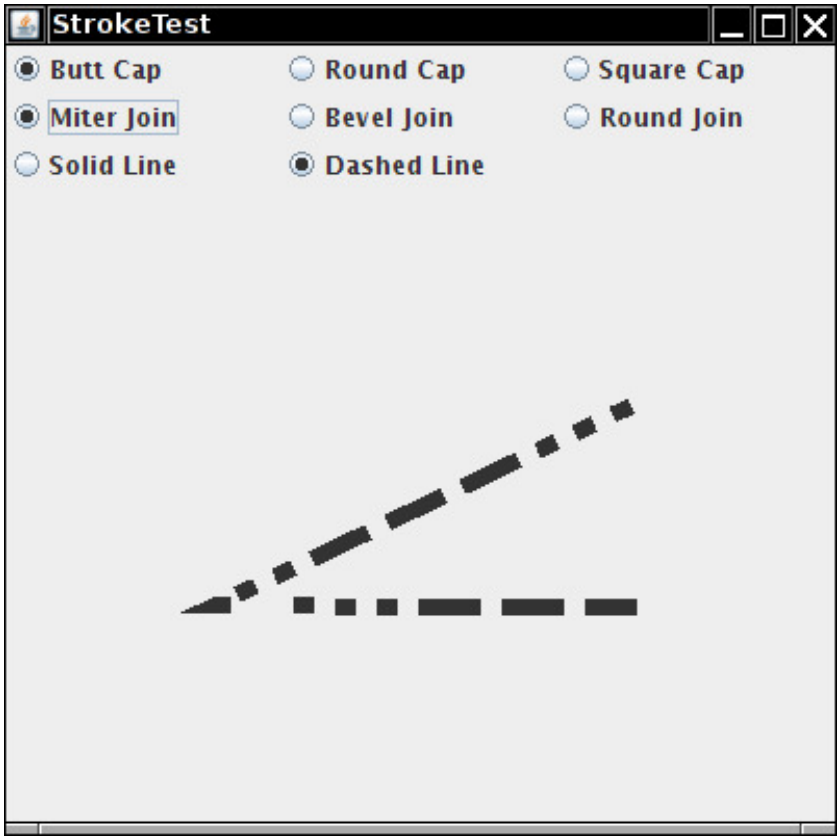


Figure 12.44: The StrokeDemo program

The program is similar to the program in Listing 12.15. The mouse listener remembers your click on the end point of a line segment, and the mouse motion listener monitors the dragging of the end point. A set of radio buttons signal the user choices for the end cap style, join style, and solid or dashed line. The paintComponent method of the StrokePanel class constructs a GeneralPath consisting of the two line segments that join the three points which the user can move with the mouse. It then constructs a BasicStroke, according to the selections the user made, and finally draws the path.

Listing 12.16 **v2ch12/stroke/StrokeDemo.java**

```
package v2ch12.stroke;

import module java.desktop;
import java.awt.event.MouseEvent;

/**
 * This program demonstrates different stroke types.
 */
class StrokeDemo {
   void main() {
      EventQueue.invokeLater(() -> {
         var frame = new StrokeTestFrame();
         frame.setTitle("StrokeTest");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         frame.setVisible(true);
      });
   }
}

/**
 * This frame lets the user choose the cap, join, and line style, and shows the re
 * stroke.
 */
class StrokeTestFrame extends JFrame {
   private StrokeComponent canvas;
   private JPanel buttonPanel;

   public StrokeTestFrame() {
      canvas = new StrokeComponent();
      add(canvas, BorderLayout.CENTER);

      buttonPanel = new JPanel();
      buttonPanel.setLayout(new GridLayout(3, 3));
      add(buttonPanel, BorderLayout.NORTH);

      var group1 = new ButtonGroup();
      makeCapButton("Butt Cap", BasicStroke.CAP_BUTT, group1);
      makeCapButton("Round Cap", BasicStroke.CAP_ROUND, group1);
      makeCapButton("Square Cap", BasicStroke.CAP_SQUARE, group1);

      var group2 = new ButtonGroup();
      makeJoinButton("Miter Join", BasicStroke.JOIN_MITER, group2);
      makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL, group2);
      makeJoinButton("Round Join", BasicStroke.JOIN_ROUND, group2);
```

```
    var group3 = new ButtonGroup();
    makeDashButton("Solid Line", false, group3);
    makeDashButton("Dashed Line", true, group3);
}

/**
 * Makes a radio button to change the cap style.
 * @param label the button label
 * @param style the cap style
 * @param group the radio button group
 */
private void makeCapButton(String label, final int style, ButtonGroup group
    // select first button in group
    boolean selected = group.getButtonCount() == 0;
    var button = new JRadioButton(label, selected);
    buttonPanel.add(button);
    group.add(button);
    button.addActionListener(_ -> canvas.setCap(style));
    pack();
}

/**
 * Makes a radio button to change the join style.
 * @param label the button label
 * @param style the join style
 * @param group the radio button group
 */
private void makeJoinButton(String label, final int style, ButtonGroup group
    // select first button in group
    boolean selected = group.getButtonCount() == 0;
    var button = new JRadioButton(label, selected);
    buttonPanel.add(button);
    group.add(button);
    button.addActionListener(_ -> canvas.setJoin(style));
}

/**
 * Makes a radio button to set solid or dashed lines.
 * @param label the button label
 * @param style false for solid, true for dashed lines
 * @param group the radio button group
 */
private void makeDashButton(String label, final boolean style, ButtonGroup
```

```
      // select first button in group
      boolean selected = group.getButtonCount() == 0;
      var button = new JRadioButton(label, selected);
      buttonPanel.add(button);
      group.add(button);
      button.addActionListener(_ -> canvas.setDash(style));
   }
}

/**
 * This component draws two joined lines, using different stroke objects, and a
 * user to drag the three points defining the lines.
 */
class StrokeComponent extends JComponent {
   private static final Dimension PREFERRED_SIZE = new Dimension(400, 400);
   private static int SIZE = 10;

   private Point2D[] points;
   private int current;
   private float width;
   private int cap;
   private int join;
   private boolean dash;

   public StrokeComponent() {
      addMouseListener(new MouseAdapter() {
         public void mousePressed(MouseEvent event) {
            Point p = event.getPoint();
            for (int i = 0; i < points.length; i++) {
               double x = points[i].getX() - SIZE / 2;
               double y = points[i].getY() - SIZE / 2;
               var r = new Rectangle2D.Double(x, y, SIZE, SIZE);
               if (r.contains(p)) {
                  current = i;
                  return;
               }
            }
         }

         public void mouseReleased(MouseEvent event) {
            current = -1;
         }
      });
```

```
      addMouseMotionListener(new MouseMotionAdapter() {
         public void mouseDragged(MouseEvent event) {
            if (current == -1) return;
            points[current] = event.getPoint();
            repaint();
         }
      });

      points = new Point2D[3];
      points[0] = new Point2D.Double(200, 100);
      points[1] = new Point2D.Double(100, 200);
      points[2] = new Point2D.Double(200, 200);
      current = -1;
      width = 8.0F;
   }

   public void paintComponent(Graphics g) {
      var g2 = (Graphics2D) g;
      var path = new GeneralPath();
      path.moveTo((float) points[0].getX(), (float) points[0].getY());
      for (int i = 1; i < points.length; i++)
         path.lineTo((float) points[i].getX(), (float) points[i].getY());
      BasicStroke stroke;
      if (dash) {
         float miterLimit = 10.0F;
         float[] dashPattern = { 10F, 10F, 10F, 10F, 10F, 10F, 30F, 10F, 30F, 10F, 30F, 1
               10F, 10F, 10F, 10F, 10F, 30F };
         float dashPhase = 0;
         stroke = new BasicStroke(width, cap, join, miterLimit, dashPattern, dash
      }
      else
         stroke = new BasicStroke(width, cap, join);
      g2.setStroke(stroke);
      g2.draw(path);
   }

   /**
    * Sets the join style.
    * @param j the join style
    */
   public void setJoin(int j) {
      join = j;
      repaint();
   }
```

```
   /**
    * Sets the cap style.
    * @param c the cap style
    */
   public void setCap(int c) {
      cap = c;
      repaint();
   }

   /**
    * Sets solid or dashed lines.
    * @param d false for solid, true for dashed lines
    */
   public void setDash(boolean d) {
      dash = d;
      repaint();
   }

   public Dimension getPreferredSize() {
      return PREFERRED_SIZE;
   }
}
```

java.awt.Graphics2D1.2

- void setStroke(Stroke s)
  sets the stroke of this graphics context to the given object that implements the Stroke interface.

java.awt.BasicStroke1.2

- BasicStroke(float width)
- BasicStroke(float width, int cap, int join)
- BasicStroke(float width, int cap, int join, float miterlimit)
- BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dashPhase)
  construct a stroke object with the given attributes.
  The end cap style is one of CAP_BUTT, CAP_ROUND, and CAP_SQUARE.
  The join style is one of JOIN_BEVEL, JOIN_MITER, and

JOIN_ROUND.

12.5.6. Paint

When you fill a shape, its inside is covered with paint. Use the setPaint method to set the paint style to an object whose class implements the Paint interface. The Java 2D API provides three such classes:

- The Color class implements the Paint interface. To fill shapes with a solid color, simply call setPaint with a Color object, such as

  ```
  g2.setPaint(Color.red);
  ```

- The GradientPaint class varies colors by interpolating between two given color values (see Figure 12.45).



Figure 12.45: Gradient paint

- The TexturePaint class fills an area with repetitions of an

image (see Figure 12.46).

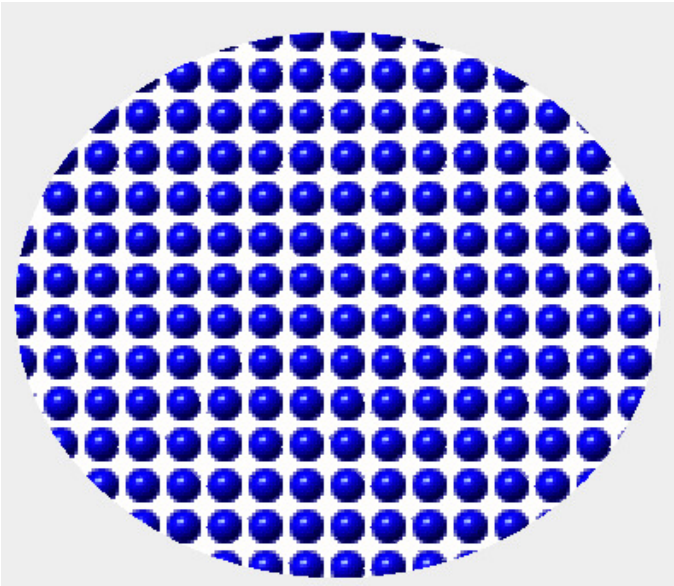


Figure 12.46: Texture paint

You can construct a GradientPaint object by specifying two points and the colors that you want at these two points.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW));
```

Colors are interpolated along the line joining the two points. Colors are constant along lines perpendicular to that joining line. Points beyond an end point of the line are given the color at the end point.

Alternatively, if you call the GradientPaint constructor with true for the cyclic parameter:

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW, true
```

Then, the color variation cycles and keeps varying beyond the end points.

To construct a TexturePaint object, specify a BufferedImage and

an anchor rectangle.

```
g2.setPaint(new TexturePaint(bufferedImage, anchorRectangle));
```

I will introduce the BufferedImage class later in this chapter when images are discussed in detail. The simplest way of obtaining a buffered image is to read an image file:

```
bufferedImage = ImageIO.read(new File("blue-ball.gif"));
```

The anchor rectangle is extended indefinitely in x and y directions to tile the entire coordinate plane. The image is scaled to fit into the anchor and then replicated into each tile.

java.awt.Graphics2D1.2

- void setPaint(Paint s)
  sets the paint of this graphics context to the given object that implements the Paint interface.

java.awt.GradientPaint1.2

- GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2)
- GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2, boolean cyclic)
- GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2)
- GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2, boolean cyclic)
  construct a gradient paint object that fills shapes with color such that the start point is colored with color1, the end point is colored with color2, and the colors in between are linearly interpolated. Colors are constant along lines perpendicular to the line joining the start and the end point. By default, the gradient paint is not cyclic—that is, points beyond the start and end points are colored with the same color as the start and end point. If the gradient

paint is cyclic, then colors continue to be interpolated, first returning to the starting point color and then repeating indefinitely in both directions.

java.awt.TexturePaint1.2

- TexturePaint(BufferedImage texture, Rectangle2D anchor) creates a texture paint object. The anchor rectangle defines the tiling of the space to be painted; it is repeated indefinitely in x and y directions, and the texture image is scaled to fill each tile.

12.5.7. Coordinate Transformations

Suppose you need to draw an object, such as an automobile. You know, from the manufacturer's specifications, the height, wheelbase, and total length. You could, of course, figure out all pixel positions, assuming some number of pixels per meter. However, there is an easier way: You can ask the graphics context to carry out the conversion for you.

```
g2.scale(pixelsPerMeter, pixelsPerMeter);
g2.draw(new Line2D.Double(coordinates in meters)); // converts to p
                                                   // draws scaled line
```

The scale method of the Graphics2D class sets the coordinate transformation of the graphics context to a scaling transformation. That transformation changes user coordinates (user-specified units) to device coordinates (pixels). Figure 12.47 shows how the transformation works.

.

Figure 12.47: User and device coordinates

Coordinate transformations are very useful in practice. They allow you to work with convenient coordinate values. The

graphics context takes care of the dirty work of transforming them to pixels.

There are four fundamental transformations.

- Scaling: blowing up, or shrinking, all distances from a fixed point
- Rotation: rotating all points around a fixed center
- Translation: moving all points by a fixed amount
- Shear: leaving one line fixed and "sliding" the lines parallel to it by an amount that is proportional to the distance from the fixed line

Figure 12.48 shows how these four fundamental transformations act on a unit square.

.

Figure 12.48: The fundamental transformations

The scale, rotate, translate, and shear methods of the Graphics2D class set the coordinate transformation of the graphics context to one of these fundamental transformations.

You can compose the transformations. For example, you might want to rotate shapes and double their size; supply both a rotation and a scaling transformation:

```
g2.rotate(angle);
g2.scale(2, 2);
g2.draw(. . .);
```

In this case, it does not matter in which order you supply the transformations. However, with most transformations, order does matter. For example, if you want to rotate and shear, then it makes a difference which of the transformations you supply first. You need to figure out what your intention is. The graphics context will apply the transformations in the order opposite to that in which you supplied them—that is, the last transformation you supply is applied first.

You can supply as many transformations as you like. For example, consider the following sequence of transformations:

```
g2.translate(x, y);
g2.rotate(a);
g2.translate(-x, -y);
```

The last transformation (which is applied first) moves the point (x, y) to the origin. The second transformation rotates with an angle a around the origin. The final transformation moves the origin back to (x, y). The overall effect is a rotation with center point (x, y)—see Figure 12.49. Since rotating about a point other than the origin is such a common operation, there is a shortcut:

```
g2.rotate(a, x, y);
```

.

Figure 12.49: Composing transformations

If you know some matrix theory, you are probably aware that all rotations, translations, scalings, shears, and their compositions can be expressed by transformation matrices of the form:

$$[\ x_{new}\ y_{new}\ 1\ ] = [\ a\ c\ e\ b\ d\ f\ 0\ 0\ 1\ ] \cdot [\ x\ y\ 1\ ]$$

Such a transformation is called an affine transformation. In the Java 2D API, the AffineTransform class describes such a transformation. If you know the components of a particular transformation matrix, you can construct it directly as

```
var t = new AffineTransform(a, b, c, d, e, f);
```

Additionally, the factory methods getRotateInstance, getScaleInstance, getTranslateInstance, and getShearInstance construct the matrices that represent these transformation types. For example, the call

```
t = AffineTransform.getScaleInstance(2.0F, 0.5F);
```

returns a transformation that corresponds to the matrix

[2 0 0 0 0.5 0 0 0 1]

Finally, the instance methods setToRotation, setToScale, setToTranslation, and setToShear set a transformation object to a new type. Here is an example:

```
t.setToRotation(angle); // sets t to a rotation
```

You can set the coordinate transformation of the graphics context to an AffineTransform object.

```
g2.setTransform(t); // replaces current transformation
```

However, in practice, you shouldn't call the setTransform operation, as it replaces any existing transformation that the graphics context may have. For example, a graphics context for printing in landscape mode already contains a 90-degree rotation transformation. If you call setTransform, you obliterate that rotation. Instead, call the transform method.

```
g2.transform(t); // composes current transformation with t
```

It composes the existing transformation with the new AffineTransform object.

If you just want to apply a transformation temporarily, first get the old transformation, compose it with your new transformation, and finally restore the old transformation when you are done.

```
AffineTransform oldTransform = g2.getTransform(); // save old trans
g2.transform(t); // apply temporary transform
draw on g2
g2.setTransform(oldTransform); // restore old transform
```

java.awt.geom.AffineTransform 1.2
■ AffineTransform(double a, double b, double c, double d, double e, double f)

- AffineTransform(float a, float b, float c, float d, float e, float f)
  construct the affine transform with matrix
  [acebdf001]
- AffineTransform(double[] m)
- AffineTransform(float[] m)
  construct the affine transform with matrix
  [m[0]m[2]m[4]m[1]m[3]m[5]001]
- static AffineTransform getRotateInstance(double a)
  creates a rotation around the origin by the angle a (in radians). The transformation matrix is
  [cos(a)-sin(a)0sin(a)cos(a)0001]
  If a is between 0 and π / 2, the rotation moves the positive x axis toward the positive y axis.
- static AffineTransform getRotateInstance(double a, double x, double y)
  creates a rotation around the point (x,y) by the angle a (in radians).
- static AffineTransform getScaleInstance(double sx, double sy)
  creates a scaling transformation that scales the x axis by sx and the y axis by sy. The transformation matrix is
  [sx000sy0001]
- static AffineTransform getShearInstance(double shx, double shy)
  creates a shear transformation that shears the x axis by shx and the y axis by shy. The transformation matrix is
  [1shx0shy10001]
- static AffineTransform getTranslateInstance(double tx, double ty)
  creates a translation that moves the x axis by tx and the y axis by ty. The transformation matrix is
  [10tx01ty001]
- void setToRotation(double a)
- void setToRotation(double a, double x, double y)
- void setToScale(double sx, double sy)
- void setToShear(double sx, double sy)
- void setToTranslation(double tx, double ty)
  set this affine transformation to a basic transformation with the given parameters. See the getXxxInstance

methods for an explanation of the basic transformations and their parameters.

java.awt.Graphics2D1.2

- void setTransform(AffineTransform t)
  replaces the existing coordinate transformation of this graphics context with t.
- void transform(AffineTransform t)
  composes the existing coordinate transformation of this graphics context with t.
- void rotate(double a)
- void rotate(double a, double x, double y)
- void scale(double sx, double sy)
- void shear(double sx, double sy)
- void translate(double tx, double ty)
  compose the existing coordinate transformation of this graphics context with a basic transformation with the given parameters. See the AffineTransform.getXxxInstance methods for an explanation of the basic transformations and their parameters.

12.5.8. Clipping

By setting a clipping shape in the graphics context, you constrain all drawing operations to the interior of that clipping shape.

```
g2.setClip(clipShape); // but see below
g2.draw(shape); // draws only the part that falls inside the clipping sh
```

However, in practice, you don't want to call the setClip operation because it replaces any existing clipping shape that the graphics context might have. For example, as you will see later in this chapter, a graphics context for printing comes with a clip rectangle that ensures that you don't draw on the

margins. Instead, call the clip method.

```
g2.clip(clipShape); // better
```

The clip method intersects the existing clipping shape with the new one that you supply.

If you just want to apply a clipping area temporarily, you should first get the old clip, add your new clip, and finally restore the old clip when you are done:

```
Shape oldClip = g2.getClip(); // save old clip
g2.clip(clipShape); // apply temporary clip
draw on g2
g2.setClip(oldClip); // restore old clip
```

In Figure 12.50, we show off the clipping capability with a rather dramatic drawing of a line pattern clipped by a complex shape—namely, the outline of a set of letters.



Figure 12.50: Using letter shapes to clip a line pattern

To obtain the character outlines, you need a font render context. Use the getFontRenderContext method of the Graphics2D class.

```
FontRenderContext context = g2.getFontRenderContext();
```

Next, using a string, a font, and the font render context, create a TextLayout object:

```
var layout = new TextLayout("Hello", font, context);
```

This text layout object describes the layout of a sequence of characters, as rendered by a particular font render context. The layout depends on the font render context—the same characters will look different on a screen and on a printer.

More important for our application, the getOutline method returns a Shape object that describes the shape of the outline of the characters in the text layout. The outline shape starts at the origin (0, 0), which might not be what you want. In that case, supply an affine transform to the getOutline operation to specify where you would like the outline to appear.

```
AffineTransform transform = AffineTransform.getTranslateInstance
Shape outline = layout.getOutline(transform);
```

Then, append the outline to the clipping shape.

```
var clipShape = new GeneralPath();
clipShape.append(outline, false);
```

Finally, set the clipping shape and draw a set of lines. The lines appear only inside the character boundaries.

```
g2.setClip(clipShape);
var p = new Point2D.Double(0, 0);
for (int i = 0; i < NLINES; i++) {
   double x = . . .;
   double y = . . .;
   var q = new Point2D.Double(x, y);
   g2.draw(new Line2D.Double(p, q)); // lines are clipped
}
```

java.awt.Graphics1.0

- void setClip(Shape s)1.2
  sets the current clipping shape to the shape s.
- Shape getClip()1.2
  returns the current clipping shape.

java.awt.Graphics2D1.2

- void clip(Shape s)
  intersects the current clipping shape with the shape s.
- FontRenderContext getFontRenderContext()
  returns a font render context that is necessary for constructing TextLayout objects.

java.awt.font.TextLayout1.2

- TextLayout(String s, Font f, FontRenderContext context)
  constructs a text layout object from a given string and font, using the font render context to obtain font properties for a particular device.
- float getAdvance()
  returns the width of this text layout.
- float getAscent()
- float getDescent()
  return the height of this text layout above and below the baseline.
- float getLeading()
  returns the distance between successive lines in the font used by this text layout.

12.5.9. Transparency and Composition

In the standard RGB color model, every color is described by its red, green, and blue components. However, it is also convenient to describe areas of an image that are transparent or partially transparent. When you superimpose an image onto an existing drawing, the transparent pixels do not obscure the pixels under

them at all, whereas partially transparent pixels are mixed with the pixels under them. Figure 12.51 shows the effect of overlaying a partially transparent rectangle on an image. You can still see the details of the image shine through from under the rectangle.



Figure 12.51: Overlaying a partially transparent rectangle on an image

In the Java 2D API, transparency is described by an alpha channel. Each pixel has, in addition to its red, green, and blue color components, an alpha value between 0 (fully transparent) and 1 (fully opaque). For example, the rectangle in Figure 12.51 was filled with a pale yellow color with 50% transparency:

```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```

Now let us look at what happens if you superimpose two shapes. You need to blend or compose the colors and alpha values of the source and destination pixels. Porter and Duff, two researchers in the field of computer graphics, have formulated 12 possible composition rules for this blending process. The Java 2D API implements all of these rules. Before going any further, I'd like to point out that only two of these rules have practical significance. If you find the rules arcane or confusing, just use the SRC_OVER rule. It is the default rule for a Graphics2D object, and it gives the most intuitive results.

Here is the theory behind the rules. Suppose you have a source pixel with alpha value a_S. In the image, there is already a

destination pixel with alpha value a_D. You want to compose the two. The diagram in Figure 12.52 shows how to design a composition rule.

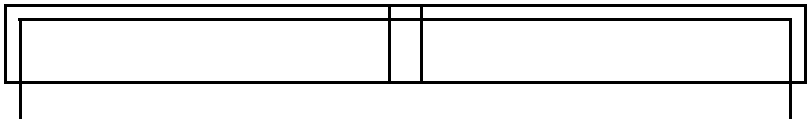
.

Figure 12.52: Designing a composition rule

Porter and Duff consider the alpha value as the probability that the pixel color should be used. From the perspective of the source, there is a probability a_S that it wants to use the source color and a probability of $1 - a_S$ that it doesn't care. The same holds for the destination. When composing the colors, let us assume that the probabilities are independent. Then there are four cases, as shown in Figure 12.52. If the source wants to use the source color and the destination doesn't care, then it seems reasonable to let the source have its way. That's why the upper right corner of the diagram is labeled "S". The probability for that event is $a_S \cdot (1 - a_D)$. Similarly, the lower left corner is labeled "D". What should one do if both destination and source would like to select their color? That's where the Porter–Duff rules come in. If we decide that the source is more important, we label the lower right corner with an "S" as well. That rule is called SRC_OVER. In that rule, you combine the source colors with a weight of a_S and the destination colors with a weight of $(1 - a_S) \cdot a_D$.

The visual effect is a blending of the source and destination, with preference given to the source. In particular, if a_S is 1, then the destination color is not taken into account at all. If a_S is 0, then the source pixel is completely transparent and the destination color is unchanged.

The other rules depend on what letters you put in the boxes of the probability diagram. Table 12.3 and Figure 12.53 show all rules that are supported by the Java 2D API. The images in the figure show the results of the rules when a rectangular source region with an alpha of 0.75 is combined with an elliptical destination region with an alpha of 1.0.



Rule	Explanation
CLEAR	Clears destination.
SRC	Source overwrites destination and empty pixels.
DST	Source does not affect destination.
SRC_OVER	Blends with destination and overwrites empty pixels.
DST_OVER	Source does not affect destination and overwrites empty pixels.
SRC_IN	Source overwrites destination.
SRC_OUT	Clears destination and overwrites empty pixels.
DST_IN	Source alpha modifies destination.
DST_OUT	Source alpha complement modifies destination.
SRC_ATOP	Blends with destination.
DST_ATOP	Source alpha modifies destination. Source overwrites empty pixels.
XOR	Source alpha complement modifies destination. Source overwrites empty pixels.

Table 12.3: The Porter–Duff Composition Rules

.

Figure 12.53: Porter–Duff composition rules

As you can see, most of the rules aren't very useful. Consider, as an extreme case, the DST_IN rule. It doesn't take the source color into account at all, but uses the alpha of the source to affect the destination. The SRC rule is potentially useful—it forces the source color to be used, turning off blending with the destination.

Use the setComposite method of the Graphics2D class to install an object of a class that implements the Composite interface. The Java 2D API supplies one such class, AlphaComposite, that implements all the Porter–Duff rules in Figure 12.53.

The factory method getInstance of the AlphaComposite class yields an AlphaComposite object. You supply the rule and the

alpha value to be used for source pixels. For example, consider the following code:

```
int rule = AlphaComposite.SRC_OVER;
float alpha = 0.5f;
g2.setComposite(AlphaComposite.getInstance(rule, alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle);
```

The rectangle is then painted with blue color and an alpha value of 0.5. Since the composition rule is SRC_OVER, it is transparently overlaid on the existing image.

The program in Listing 12.17 lets you explore these composition rules. Pick a rule from the combo box and use the slider to set the alpha value of the AlphaComposite object.

Furthermore, the program displays a verbal description of each rule. Note that the descriptions are computed from the composition rule diagrams. For example, a "DS" in the second row stands for “blends with destination.”

The program has one important twist. There is no guarantee that the graphics context that corresponds to the screen has an alpha channel. (In fact, it generally does not.) When pixels are deposited to a destination without an alpha channel, the pixel colors are multiplied with the alpha value and the alpha value is discarded. Now, several of the Porter–Duff rules use the alpha values of the destination, which means a destination alpha channel is important. For that reason, we use a buffered image with the ARGB color model to compose the shapes. After the images have been composed, we draw the resulting image to the screen.

```
var image = new BufferedImage(getWidth(), getHeight(), BufferedIm
Graphics2D gImage = image.createGraphics();
// now draw to gImage
g2.drawImage(image, null, 0, 0);
```

Listing 12.17 shows the frame class. The component class is in Listing 12.18. The Rule class in Listing 12.19 provides a brief

explanation for each rule—see Figure 12.54. As you run the program, move the alpha slider from left to right to see the effect on the composed shapes. In particular, note that the only difference between the DST_IN and DST_OUT rules is how the destination (!) color changes when you change the source alpha.

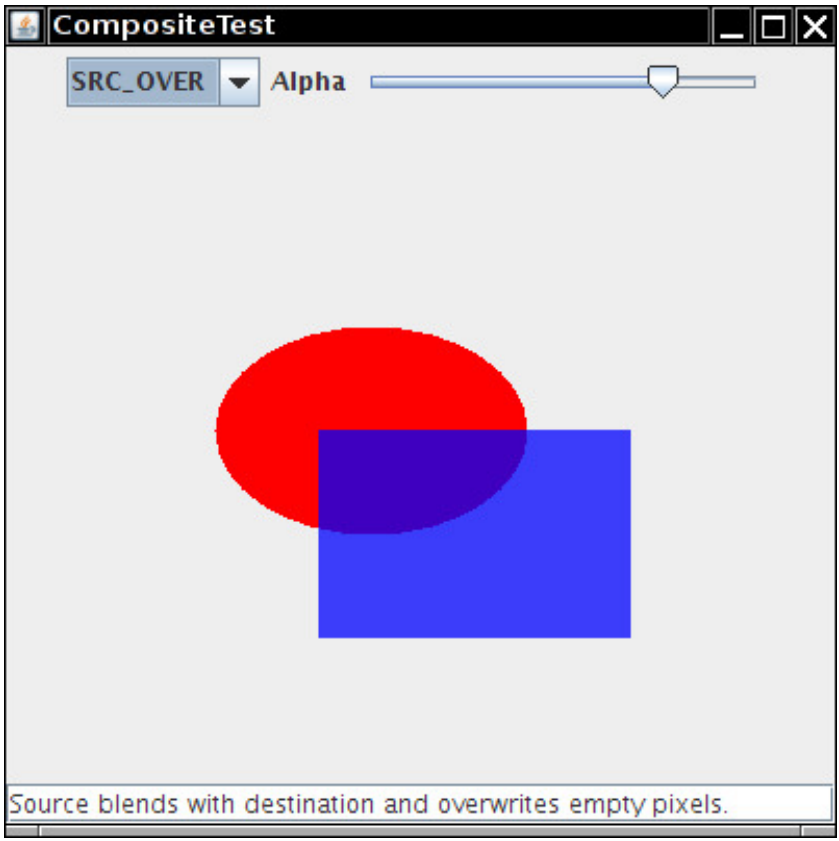


Figure 12.54: The CompositeDemo program

Listing 12.17 **v2ch12/composite/CompositeDemoFrame.java**

```
package v2ch12.composite;
```

```
import module java.desktop;

/**
 * This frame contains a combo box to choose a composition rule, a slider to ch
 * alpha channel, and a component that shows the composition.
 */
class CompositeDemoFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 400;
   private static final int DEFAULT_HEIGHT = 400;

   private CompositeComponent canvas;
   private JComboBox<Rule> ruleCombo;
   private JSlider alphaSlider;
   private JTextField explanation;

   public CompositeDemoFrame() {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      canvas = new CompositeComponent();
      add(canvas, BorderLayout.CENTER);

      ruleCombo = new JComboBox<>(
            new Rule[] { new Rule("CLEAR", "  ", "  "), new Rule("SRC", " S", " S"),
                  new Rule("DST", "  ", "DD"), new Rule("SRC_OVER", " S", "DS"),
                  new Rule("DST_OVER", " S", "DD"), new Rule("SRC_IN", "  ", " S"),
                  new Rule("SRC_OUT", " S", "  "), new Rule("DST_IN", "  ", " D"),
                  new Rule("DST_OUT", "  ", "D "), new Rule("SRC_ATOP", "  ", "DS"),
                  new Rule("DST_ATOP", " S", " D"), new Rule("XOR", " S", "D "), });
      ruleCombo.addActionListener(_ -> {
         var r = (Rule) ruleCombo.getSelectedItem();
         canvas.setRule(r.getValue());
         explanation.setText(r.getExplanation());
      });

      alphaSlider = new JSlider(0, 100, 75);
      alphaSlider.addChangeListener(_ -> canvas.setAlpha(alphaSlider.getValue(
      var panel = new JPanel();
      panel.add(ruleCombo);
      panel.add(new JLabel("Alpha"));
      panel.add(alphaSlider);
      add(panel, BorderLayout.NORTH);

      explanation = new JTextField();
      add(explanation, BorderLayout.SOUTH);
```

```
      canvas.setAlpha(alphaSlider.getValue());
      Rule r = ruleCombo.getItemAt(ruleCombo.getSelectedIndex());
      canvas.setRule(r.getValue());
      explanation.setText(r.getExplanation());
   }
}
```

Listing 12.18 **v2ch12/composite/CompositeComponent.java**

```
package v2ch12.composite;

import module java.desktop;

/**
 * This component draws two shapes, composed with a composition rule.
 */
class CompositeComponent extends JComponent {
   private int rule;
   private Shape shape1;
   private Shape shape2;
   private float alpha;

   public CompositeComponent() {
      shape1 = new Ellipse2D.Double(100, 100, 150, 100);
      shape2 = new Rectangle2D.Double(150, 150, 150, 100);
   }

   public void paintComponent(Graphics g) {
      var g2 = (Graphics2D) g;

      var image = new BufferedImage(getWidth(), getHeight(), BufferedImage.T
      Graphics2D gImage = image.createGraphics();
      gImage.setPaint(Color.red);
      gImage.fill(shape1);
      AlphaComposite composite = AlphaComposite.getInstance(rule, alpha);
      gImage.setComposite(composite);
      gImage.setPaint(Color.blue);
      gImage.fill(shape2);
      g2.drawImage(image, null, 0, 0);
   }

   /**
```

```
    * Sets the composition rule.
    * @param r the rule (as an AlphaComposite constant)
    */
   public void setRule(int r) {
      rule = r;
      repaint();
   }

   /**
    * Sets the alpha of the source.
    * @param a the alpha value between 0 and 100
    */
   public void setAlpha(int a) {
      alpha = (float) a / 100.0F;
      repaint();
   }
}
```

Listing 12.19 **v2ch12/composite/Rule.java**

```
package v2ch12.composite;

import module java.desktop;

/**
 * This class describes a Porter-Duff rule.
 */
public class Rule {
   private String name;
   private String porterDuff1;
   private String porterDuff2;

   /**
    * Constructs a Porter-Duff rule.
    * @param n the rule name
    * @param pd1 the first row of the Porter-Duff square
    * @param pd2 the second row of the Porter-Duff square
    */
   public Rule(String n, String pd1, String pd2) {
      name = n;
      porterDuff1 = pd1;
      porterDuff2 = pd2;
   }
```

```
   /**
    * Gets an explanation of the behavior of this rule.
    * @return the explanation
    */
   public String getExplanation() {
      var r = new StringBuilder("Source ");
      if (porterDuff2.equals("  ")) r.append("clears");
      if (porterDuff2.equals(" S")) r.append("overwrites");
      if (porterDuff2.equals("DS")) r.append("blends with");
      if (porterDuff2.equals(" D")) r.append("alpha modifies");
      if (porterDuff2.equals("D ")) r.append("alpha complement modifies");
      if (porterDuff2.equals("DD")) r.append("does not affect");
      r.append(" destination");
      if (porterDuff1.equals(" S")) r.append(" and overwrites empty pixels");
      r.append(".");
      return r.toString();
   }

   public String toString() {
      return name;
   }

   /**
    * Gets the value of this rule in the AlphaComposite class.
    * @return the AlphaComposite constant value, or -1 if there is no matching
    */
   public int getValue() {
      try {
         return (Integer) AlphaComposite.class.getField(name).get(null);
      }
      catch (Exception e) {
         return -1;
      }
   }
}
```

java.awt.Graphics2D 1.2

- void setComposite(Composite s)
  sets the composite of this graphics context to the given object that implements the Composite interface.

java.awt.AlphaComposite 1.2

- static AlphaComposite getInstance(int rule)
- static AlphaComposite getInstance(int rule, float sourceAlpha)
  construct an alpha composite object. The rule is one of CLEAR, SRC, SRC_OVER, DST_OVER, SRC_IN, SRC_OUT, DST_IN, DST_OUT, DST, DST_ATOP, SRC_ATOP, XOR.

12.6. Raster Images

The Java2D API lets you create drawings that are made up of lines, curves, and areas. It is a "vector" API because you specify the mathematical properties of the shapes. However, for processing images that are made up of pixels, you want to work with a "raster" of color data. The following sections show you how to process raster images in Java.

12.6.1. Readers and Writers for Images

The javax.imageio package contains out-of-the-box support for reading and writing several common file formats, as well as a framework that enables third parties to add readers and writers for other formats. The GIF, JPEG, PNG, BMP (Windows bitmap), and WBMP (wireless bitmap) file formats are supported.

The basics of the library are extremely straightforward. To load an image, use the static read method of the ImageIO class:

```
File f = . . .;
BufferedImage image = ImageIO.read(f);
```

The ImageIO class picks an appropriate reader, based on the file type. It may consult the file extension and the "magic number" at the beginning of the file for that purpose. If no suitable reader can be found or the reader can't decode the file contents, the read method returns null.

Writing an image to a file is just as simple:

```
File f = . . .;
String format = . . .;
ImageIO.write(image, format, f);
```

Here the format string is a string identifying the image format, such as "JPEG" or "PNG". The ImageIO class picks an appropriate writer and saves the file.

12.6.2. Obtaining Readers and Writers for Image File Types

For more advanced image reading and writing operations that go beyond the static read and write methods of the ImageIO class, you first need to get the appropriate ImageReader and ImageWriter objects. The ImageIO class enumerates readers and writers that match one of the following:

- An image format (such as "JPEG")
- A file suffix (such as "jpg")
- A MIME type (such as "image/jpeg")

> MIME is the Multipurpose Internet Mail Extensions standard. The MIME standard defines common data formats such as image/jpeg and application/pdf.

For example, you can obtain a reader that reads JPEG files as follows:

```
ImageReader reader = null;
Iterator<ImageReader> iter = ImageIO.getImageReadersByFormatNa
if (iter.hasNext()) reader = iter.next();
```

The getImageReadersBySuffix and getImageReadersByMIMEType methods enumerate readers that match a file extension or MIME type.

It is possible that the ImageIO class can locate multiple readers that can all read a particular image type. In that case, you have

to pick one of them, but it isn't clear how you can decide which one is the best. To find out more information about a reader, obtain its service provider interface:

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Then you can get the vendor name and version number:

```
String vendor = spi.getVendor();
String version = spi.getVersion();
```

Perhaps that information can help you decide among the choices—or you might just present a list of readers to your program users and let them choose. For now, we assume that the first enumerated reader is adequate.

In the sample program in Listing 12.20, we want to find all file suffixes of all available readers so that we can use them in a file filter. Use the static ImageIO.getReaderFileSuffixes method for this purpose:

```
String[] extensions = ImageIO.getWriterFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter("Image files", ext
```

For saving files, we have to work harder. We'd like to present the user with a menu of all supported image types. Unfortunately, the getWriterFormatNames of the ImageIO class returns a rather curious list with redundant names, such as

```
jpg, BMP, bmp, JPG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gif, TIF
```

That's not something one would want to present in a menu. What is needed is a list of "preferred" format names. The helper method getWriterFormats in Listing 12.20 looks up the first writer associated with each format name, and asks it what its format names are, in the hope that it will list the most popular one first. Indeed, for the JPEG writer, this works fine—it lists "JPEG" before the other options. The PNG writer, on the other hand, lists "png" in lower case before "PNG". Therefore, for consistency, the helper method forces lowercase names to upper case.

12.6.3. Reading and Writing Files with Multiple Images

Some files—in particular, animated GIF files—contain multiple images. The read method of the ImageIO class reads a single image. To read multiple images, turn the input source (for example, an input stream or file) into an ImageInputStream.

```
InputStream in = . . .;
ImageInputStream imageIn = ImageIO.createImageInputStream(in);
```

Then, attach the image input stream to the reader:

```
reader.setInput(imageIn, true);
```

The second argument indicates that the input is in "seek forward only" mode. Otherwise, random access is used, either by buffering stream input as it is read or by using random file access. Random access is required for certain operations. For example, to find out the number of images in a GIF file, you need to read the entire file. If you then want to fetch an image, the input must be read again.

This consideration is only important if you read from a stream, if the input contains multiple images, and if the image format doesn't have the information that you request (such as the image count) in the header. If you read from a file, simply use

```
File f = . . .;
ImageInputStream imageIn = ImageIO.createImageInputStream(f);
reader.setInput(imageIn);
```

Once you have a reader, you can read the images in the input by calling

```
BufferedImage image = reader.read(index);
```

where index is the image index, starting with 0.

If the input is in the "seek forward only" mode, you keep reading images until the read method throws an

IndexOutOfBoundsException. Otherwise, you can call the getNumImages method:

```
int n = reader.getNumImages(true);
```

Here, the argument indicates that you allow a search of the input to determine the number of images. That method throws an IllegalStateException if the input is in the "seek forward only" mode. Alternatively, you can set the "allow search" parameter to false. Then the getNumImages method returns -1 if it can't determine the number of images without a search. In that case, you'll have to switch to Plan B and keep reading images until you get an IndexOutOfBoundsException.

Some files contain thumbnails—smaller versions of an image for preview purposes. You can get the number of thumbnails of an image with the call

```
int count = reader.getNumThumbnails(index);
```

Then you get a particular index as

```
BufferedImage thumbnail = reader.getThumbnail(index, thumbnailI
```

Sometimes you may want to get the image size before actually getting the image—in particular, if the image is huge or comes from a slow network connection. Use the calls

```
int width = reader.getWidth(index);
int height = reader.getHeight(index);
```

to get the dimensions of an image with a given index.

To write a file with multiple images, you first need an ImageWriter. The ImageIO class can enumerate the writers capable of writing a particular image format:

```
String format = . . .;
ImageWriter writer = null;
Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatNa
if (iter.hasNext()) writer = iter.next();
```

Next, turn an output stream or file into an ImageOutputStream and attach it to the writer. For example,

```
File f = . . .;
ImageOutputStream imageOut = ImageIO.createImageOutputStream
writer.setOutput(imageOut);
```

You must wrap each image into an IIOImage object. You can optionally supply a list of thumbnails and image metadata (such as compression algorithms and color information). In this example, we just use null for both; see the API documentation for additional information.

```
var iioImage = new IIOImage(images[i], null, null);
```

To write out the first image, use the write method:

```
writer.write(new IIOImage(images[0], null, null));
```

For subsequent images, use

```
if (writer.canInsertImage(i))
   writer.writeInsert(i, iioImage, null);
```

The third argument can contain an ImageWriteParam object to set image writing details such as tiling and compression; use null for default values.

Not all file formats can handle multiple images. In that case, the canInsertImage method returns false for i > 0, and only a single image is saved.

The program in Listing 12.20 lets you load and save files in the formats for which the Java library supplies readers and writers. The program displays multiple images (see Figure 12.55), but not thumbnails.



Figure 12.55: An animated GIF image

Listing 12.20 **v2ch12/imageIO/ImageIOFrame.java**

```
package v2ch12.imageIO;

import module java.base;
import module java.desktop;
import java.util.List;

/**
 * This frame displays the loaded images. The menu has items for loading and
 */
public class ImageIOFrame extends JFrame {
   private static final int DEFAULT_WIDTH = 400;
```

```
private static final int DEFAULT_HEIGHT = 400;

private static Set<String> writerFormats = getWriterFormats();

private BufferedImage[] images;

public ImageIOFrame() {
   setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

   var fileMenu = new JMenu("File");
   var openItem = new JMenuItem("Open");
   openItem.addActionListener(_ -> openFile());
   fileMenu.add(openItem);

   var saveMenu = new JMenu("Save");
   fileMenu.add(saveMenu);
   Iterator<String> iter = writerFormats.iterator();
   while (iter.hasNext()) {
      final String formatName = iter.next();
      var formatItem = new JMenuItem(formatName);
      saveMenu.add(formatItem);
      formatItem.addActionListener(_ -> saveFile(formatName));
   }

   var exitItem = new JMenuItem("Exit");
   exitItem.addActionListener(_ -> System.exit(0));
   fileMenu.add(exitItem);

   var menuBar = new JMenuBar();
   menuBar.add(fileMenu);
   setJMenuBar(menuBar);
}

/**
 * Open a file and load the images.
 */
public void openFile() {
   var chooser = new JFileChooser();
   chooser.setCurrentDirectory(new File("."));
   String[] extensions = ImageIO.getReaderFileSuffixes();
   chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensio
   int r = chooser.showOpenDialog(this);
   if (r != JFileChooser.APPROVE_OPTION) return;
   File f = chooser.getSelectedFile();
```

```
      Box box = Box.createVerticalBox();
      try {
         String name = f.getName();
         String suffix = name.substring(name.lastIndexOf(".") + 1);
         Iterator<ImageReader> iter = ImageIO.getImageReadersBySuffix(suffix)
         ImageReader reader = iter.next();
         ImageInputStream imageIn = ImageIO.createImageInputStream(f);
         reader.setInput(imageIn);
         int count = reader.getNumImages(true);
         images = new BufferedImage[count];
         for (int i = 0; i < count; i++) {
            images[i] = reader.read(i);
            box.add(new JLabel(new ImageIcon(images[i])));
         }
      }
      catch (IOException e) {
         JOptionPane.showMessageDialog(this, e);
      }
      setContentPane(new JScrollPane(box));
      validate();
   }

   /**
    * Save the current image in a file.
    * @param formatName the file format
    */
   public void saveFile(final String formatName) {
      if (images == null) return;
      Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(fo
      ImageWriter writer = iter.next();
      var chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));
      String[] extensions = writer.getOriginatingProvider().getFileSuffixes();
      chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensio

      int r = chooser.showSaveDialog(this);
      if (r != JFileChooser.APPROVE_OPTION) return;
      File f = chooser.getSelectedFile();
      try {
         ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
         writer.setOutput(imageOut);

         writer.write(new IIOImage(images[0], null, null));
         for (int i = 1; i < images.length; i++) {
```

```
            var iioImage = new IIOImage(images[i], null, null);
            if (writer.canInsertImage(i)) writer.writeInsert(i, iioImage, null);
         }
      }
      catch (IOException e) {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
    * Gets a set of "preferred" format names of all image writers. The preferred
    * format name is the first format name that a writer specifies.
    * @return the format name set
    */
   public static Set<String> getWriterFormats() {
      var writerFormats = new TreeSet<String>();
      var formatNames = List.of(ImageIO.getWriterFormatNames());
      while (formatNames.size() > 0) {
         String name = formatNames.iterator().next();
         Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(
         ImageWriter writer = iter.next();
         String[] names = writer.getOriginatingProvider().getFormatNames();
         String format = names[0];
         if (format.equals(format.toLowerCase())) format = format.toUpperCase(
         writerFormats.add(format);
         formatNames.removeAll(List.of(names));
      }
      return writerFormats;
   }
}
```

javax.imageio.ImageIO 1.4

- static BufferedImage read(File input)
- static BufferedImage read(InputStream input)
- static BufferedImage read(URL input)
  read an image from input.
- static boolean write(RenderedImage image, String formatName, File output)
- static boolean write(RenderedImage image, String formatName, OutputStream output)
  write an image in the given format to output. Return false if no appropriate writer was found.

- static Iterator<ImageReader> getImageReadersByFormatName(String formatName)
- static Iterator<ImageReader> getImageReadersBySuffix(String fileSuffix)
- static Iterator<ImageReader> getImageReadersByMIMEType(String mimeType)
- static Iterator<ImageWriter> getImageWritersByFormatName(String formatName)
- static Iterator<ImageWriter> getImageWritersBySuffix(String fileSuffix)
- static Iterator<ImageWriter> getImageWritersByMIMEType(String mimeType)
  get all readers and writers that are able to handle the given format (e.g., "JPEG"), file suffix (e.g., "jpg"), or MIME type (e.g., "image/jpeg").
- static String[] getReaderFormatNames()
- static String[] getReaderMIMETypes()
- static String[] getWriterFormatNames()
- static String[] getWriterMIMETypes()
- static String[] getReaderFileSuffixes()6
- static String[] getWriterFileSuffixes()6
  get all format names, MIME type names, and file suffixes supported by readers and writers.
- ImageInputStream createImageInputStream(Object input)
- ImageOutputStream createImageOutputStream(Object output)
  create an image input or image output stream from the given object. The object can be a file, a stream, a RandomAccessFile, or another object for which a service provider exists. Return null if no registered service provider can handle the object.

javax.imageio.ImageReader1.4

- void setInput(Object input)
- void setInput(Object input, boolean seekForwardOnly)
  set the input source of the reader. The input parameter is an ImageInputStream object or another object that this reader can accept. The seekForwardOnly parameter is

true if the reader should read forward only. By default, the reader uses random access and, if necessary, buffers image data.

- BufferedImage read(int index)
  reads the image with the given image index (starting at 0). Throws an IndexOutOfBoundsException if no such image is available.
- int getNumImages(boolean allowSearch)
  gets the number of images in this reader. If allowSearch is false and the number of images cannot be determined without reading forward, then -1 is returned. If allowSearch is true and the reader is in the "seek forward only" mode, then an IllegalStateException is thrown.
- int getNumThumbnails(int index)
  gets the number of thumbnails of the image with the given index.
- BufferedImage readThumbnail(int index, int thumbnailIndex)
  gets the thumbnail with index thumbnailIndex of the image with the given index.
- int getWidth(int index)
- int getHeight(int index)
  get the image width and height. Throw an IndexOutOfBoundsException if no such image is available.
- ImageReaderSpi getOriginatingProvider()
  gets the service provider that constructed this reader.

javax.imageio.spi.IIOServiceProvider1.4

- String getVendorName()
- String getVersion()
  get the vendor name and version of this service provider.

javax.imageio.spi.ImageReaderWriterSpi1.4

- String[] getFormatNames()
- String[] getFileSuffixes()

- String[] getMIMETypes()
  get the format names, file suffixes, and MIME types supported by the readers or writers that this service provider creates.

javax.imageio.ImageWriter1.4

- void setOutput(Object output)
  sets the output target of this writer. The output parameter is an ImageOutputStream object or another object that this writer can accept.
- void write(IIOImage image)
- void write(RenderedImage image)
  write a single image to the output.
- void writeInsert(int index, IIOImage image, ImageWriteParam param)
  writes an image into a multi-image file.
- boolean canInsertImage(int index)
  returns true if it is possible to insert an image at the given index.
- ImageWriterSpi getOriginatingProvider()
  gets the service provider that constructed this writer.

javax.imageio.IIOImage1.4

- IIOImage(RenderedImage image, List thumbnails, IIOMetadata metadata)
  constructs an IIOImage from an image, optional thumbnails, and optional metadata.

12.6.4. Image Manipulation

Suppose you have an image and you would like to improve its appearance. You then need to access the individual pixels of the image and replace them with other pixels. Or perhaps you want to compute the pixels of an image from scratch—for

example, to show the result of physical measurements or a mathematical computation. The BufferedImage class gives you control over the pixels in an image, and the classes that implement the BufferedImageOp interface let you transform images.

JDK 1.0 had a completely different, and far more complex, imaging framework that was optimized for incremental rendering of images downloaded from the Web, a scan line at a time. However, it was difficult to manipulate those images. We do not discuss that framework in this book.

Most of the images that you manipulate are simply read in from an image file—they were either produced by a device such as a digital camera or scanner, or constructed by a drawing program. In this section, we'll show you a different technique for constructing an image—namely, building it up a pixel at a time.

To create an image, construct a BufferedImage object in the usual way.

```
image = new BufferedImage(width, height, BufferedImage.TYPE_INT
```

Now, call the getRaster method to obtain an object of type WritableRaster. You will use this object to access and modify the pixels of the image.

```
WritableRaster raster = image.getRaster();
```

The setPixel method lets you set an individual pixel. The complexity here is that you can’t simply set the pixel to a Color value. You must know how the buffered image specifies color values. That depends on the type of the image. If your image has a type of TYPE_INT_ARGB, then each pixel is described by four values—red, green, blue, and alpha, each between 0 and 255. You have to supply them in an array of four integers:

```
int[] black = { 0, 0, 0, 255 };
```

```
raster.setPixel(i, j, black);
```

In the lingo of the Java 2D API, these values are called the sample values of the pixel.

There are also setPixel methods with parameters of types float[] and double[]. However, the values that you need to place into these arrays are not normalized color values between 0.0 and 1.0.

```
float[] red = { 1.0F, 0.0F, 0.0F, 1.0F };
raster.setPixel(i, j, red); // ERROR
```

You need to supply values between 0 and 255, no matter what the type of the array is.

You can supply a batch of pixels with the setPixels method. Specify the starting pixel position and the width and height of the rectangle that you want to set. Then, supply an array that contains the sample values for all pixels. For example, if your buffered image has a type of TYPE_INT_ARGB, supply the red, green, blue, and alpha values of the first pixel, then the red, green, blue, and alpha values for the second pixel, and so on.

```
var pixels = new int[4 * width * height];
pixels[0] = . . .; // red value for first pixel
pixels[1] = . . .; // green value for first pixel
pixels[2] = . . .; // blue value for first pixel
pixels[3] = . . .; // alpha value for first pixel
. . .
raster.setPixels(x, y, width, height, pixels);
```

Conversely, to read a pixel, use the getPixel method. Supply an array of four integers to hold the sample values.

```
var sample = new int[4];
raster.getPixel(x, y, sample);
var color = new Color(sample[0], sample[1], sample[2], sample[3]);
```

You can read multiple pixels with the getPixels method.

```
raster.getPixels(x, y, width, height, samples);
```

If you use an image type other than TYPE_INT_ARGB and you know how that type represents pixel values, you can still use the getPixel/setPixel methods. However, you have to know the encoding of the sample values in the particular image type.

If you need to manipulate an image with an arbitrary, unknown image type, then you have to work a bit harder. Every image type has a color model that can translate between sample value arrays and the standard RGB color model.

The RGB color model isn't as standard as you might think. The exact look of a color value depends on the characteristics of the imaging device. Digital cameras, scanners, monitors, and LCD displays all have their own idiosyncrasies. As a result, the same RGB value can look quite different on different devices. The International Color Consortium (https://www.color.org) recommends that all color data be accompanied by an ICC profile that specifies how the colors map to a standard form such as the 1931 CIE XYZ color specification. That specification was designed by the Commission Internationale de l'Eclairage, or CIE (https://www.cie.co.at), the international organization in charge of providing technical guidance in all matters of illumination and color. The specification is a standard method for representing any color that the human eye can perceive as a triplet of coordinates called X, Y, Z. (See, for example, Computer Graphics: Principles and Practice, Third Edition, by John F. Hughes et al., Chapter 28, for more information on the 1931 CIE XYZ specification.)

ICC profiles are complex, however. A simpler proposed standard, called sRGB (https://www.w3.org/Graphics/Color/sRGB.html), specifies an exact mapping between RGB values and the 1931 CIE XYZ values that was designed to work well with typical color monitors. The Java 2D API uses that mapping when converting between RGB and other color spaces.

The getColorModel method returns the color model:

```
ColorModel model = image.getColorModel();
```

To find the color value of a pixel, call the getDataElements method of the Raster class. That call returns an Object that contains a color-model-specific description of the color value.

```
Object data = raster.getDataElements(x, y, null);
```

> The object that is returned by the getDataElements method is actually an array of sample values. You don't need to know this to process the object, but it explains why the method is called getDataElements.

The color model can translate the object to standard ARGB values. The getRGB method returns an int value that has the alpha, red, green, and blue values packed in four blocks of eight bits each. You can construct a Color value out of that integer with the Color(int argb, boolean hasAlpha) constructor:

```
int argb = model.getRGB(data);
var color = new Color(argb, true);
```

To set a pixel to a particular color, reverse these steps. The getRGB method of the Color class yields an int value with the alpha, red, green, and blue values. Supply that value to the getDataElements method of the ColorModel class. The return value is an Object that contains the color-model-specific description of the color value. Pass the object to the setDataElements method of the WritableRaster class.

```
int argb = color.getRGB();
Object data = model.getDataElements(argb, null);
raster.setDataElements(x, y, data);
```

To illustrate how to use these methods to build an image from individual pixels, we bow to tradition and draw a Mandelbrot set, as shown in Figure 12.56.

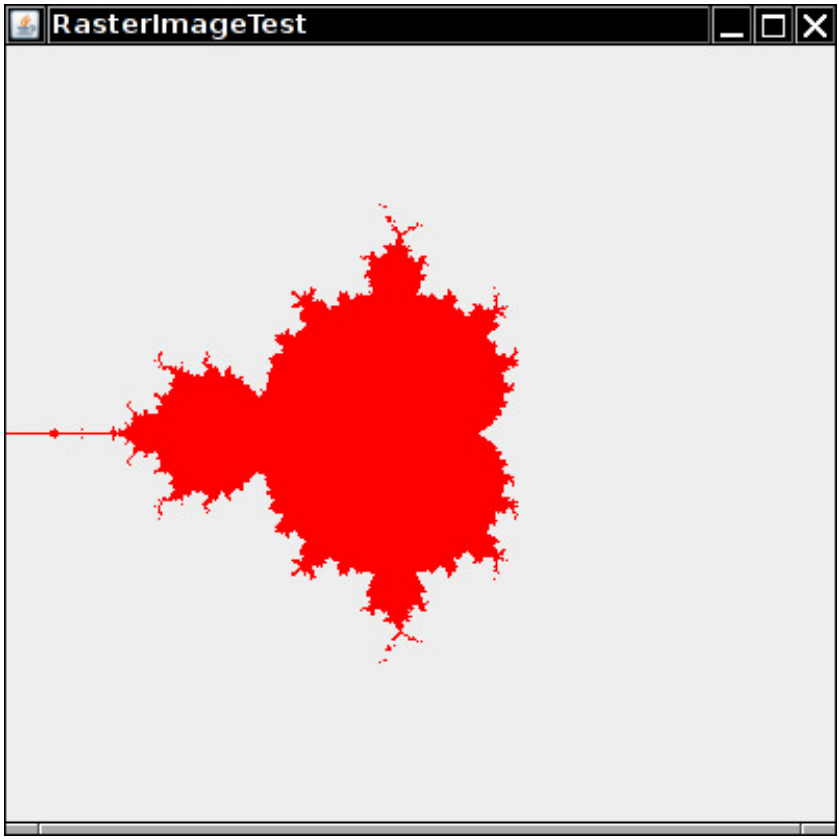


Figure 12.56: A Mandelbrot set

The idea of the Mandelbrot set is that each point of the plane is associated with a sequence of numbers. If that sequence stays bounded, you color the point. If it “escapes to infinity,” you leave it transparent.

Here is how you can construct the simplest Mandelbrot set. For each point (a, b), look at sequences that start with (x, y) = (0, 0) and iterate:

$$x_{new} = x^2 - y^2 + a$$

$$y_{new} = 2 \cdot x \cdot y + b$$

It turns out that if x or y ever gets larger than 2, then the

sequence escapes to infinity. Only the pixels that correspond to points (a, b) leading to a bounded sequence are colored. (The formulas for the number sequences come ultimately from the mathematics of complex numbers; we'll just take them for granted.)

Listing 12.21 shows the code. This program demonstrates how to use the ColorModel class for translating Color values into pixel data. That process is independent of the image type. Just for fun, change the color type of the buffered image to TYPE_BYTE_GRAY. You don't need to change any other code—the color model of the image automatically takes care of the conversion from colors to sample values.

Listing 12.21 **v2ch12/rasterImage/RasterImageFrame.java**

```
package v2ch12.rasterImage;

import module java.desktop;

/**
 * This frame shows an image with a Mandelbrot set.
 */
public class RasterImageFrame extends JFrame {
   private static final double XMIN = -2;
   private static final double XMAX = 2;
   private static final double YMIN = -2;
   private static final double YMAX = 2;
   private static final int MAX_ITERATIONS = 16;
   private static final int IMAGE_WIDTH = 400;
   private static final int IMAGE_HEIGHT = 400;

   public RasterImageFrame() {
      BufferedImage image = makeMandelbrot(IMAGE_WIDTH, IMAGE_HEIGHT
      add(new JLabel(new ImageIcon(image)));
      pack();
   }

   /**
    * Makes the Mandelbrot image.
    * @param width the width
```

```
     * @parah height the height
     * @return the image
     */
    public BufferedImage makeMandelbrot(int width, int height) {
       var image = new BufferedImage(width, height, BufferedImage.TYPE_INT_A
       WritableRaster raster = image.getRaster();
       ColorModel model = image.getColorModel();

       Color fractalColor = Color.RED;
       int argb = fractalColor.getRGB();
       Object colorData = model.getDataElements(argb, null);

       for (int i = 0; i < width; i++)
          for (int j = 0; j < height; j++) {
             double a = XMIN + i * (XMAX - XMIN) / width;
             double b = YMIN + j * (YMAX - YMIN) / height;
             if (!escapesToInfinity(a, b)) raster.setDataElements(i, j, colorData);
          }
       return image;
    }

    private boolean escapesToInfinity(double a, double b) {
       double x = 0.0;
       double y = 0.0;
       int iterations = 0;
       while (x <= 2 && y <= 2 && iterations < MAX_ITERATIONS) {
          double xnew = x * x - y * y + a;
          double ynew = 2 * x * y + b;
          x = xnew;
          y = ynew;
          iterations++;
       }
       return x > 2 || y > 2;
    }
}
```

java.awt.image.BufferedImage 1.2

- BufferedImage(int width, int height, int imageType)
  constructs a buffered image object.
  The most common image types are TYPE_INT_RGB, TYPE_INT_ARGB, TYPE_BYTE_GRAY, and TYPE_BYTE_INDEXED.

- ColorModel getColorModel()
  returns the color model of this buffered image.
- WritableRaster getRaster()
  gets the raster for accessing and modifying pixels of this buffered image.

java.awt.image.Raster1.2

- Object getDataElements(int x, int y, Object data)
  returns the sample data for a raster point, in an array whose element type and length depend on the color model. If data is not null, it is assumed to be an array that is appropriate for holding sample data, and it is filled. If data is null, a new array is allocated. Its element type and length depend on the color model.
- int[] getPixel(int x, int y, int[] sampleValues)
- float[] getPixel(int x, int y, float[] sampleValues)
- double[] getPixel(int x, int y, double[] sampleValues)
- int[] getPixels(int x, int y, int width, int height, int[] sampleValues)
- float[] getPixels(int x, int y, int width, int height, float[] sampleValues)
- double[] getPixels(int x, int y, int width, int height, double[] sampleValues)
  return the sample values for a raster point, or a rectangle of raster points, in an array whose length depends on the color model. If sampleValues is not null, it is assumed to be sufficiently long for holding the sample values, and it is filled. If sampleValues is null, a new array is allocated. These methods are only useful if you know the meaning of the sample values for a color model.

java.awt.image.WritableRaster1.2

- void setDataElements(int x, int y, Object data)
  sets the sample data for a raster point. data is an array filled with the sample data for a pixel. Its element type and

length depend on the color model.

- void setPixel(int x, int y, int[] sampleValues)
- void setPixel(int x, int y, float[] sampleValues)
- void setPixel(int x, int y, double[] sampleValues)
- void setPixels(int x, int y, int width, int height, int[] sampleValues)
- void setPixels(int x, int y, int width, int height, float[] sampleValues)
- void setPixels(int x, int y, int width, int height, double[] sampleValues)
  set the sample values for a raster point or a rectangle of raster points. These methods are only useful if you know the encoding of the sample values for a color model.

java.awt.image.ColorModel1.2

- int getRGB(Object data)
  returns the ARGB value that corresponds to the sample data passed in the data array. Its element type and length depend on the color model.
- Object getDataElements(int argb, Object data);
  returns the sample data for a color value. If data is not null, it is assumed to be an array that is appropriate for holding sample data, and it is filled. If data is null, a new array is allocated. data is an array filled with the sample data for a pixel. Its element type and length depend on the color model.

java.awt.Color1.0

- Color(int argb, boolean hasAlpha)1.2
  creates a color with the specified combined ARGB value if hasAlpha is true, or the specified RGB value if hasAlpha is false.
- int getRGB()
  returns the ARGB color value corresponding to this color.

12.6.5. Filtering Images

In the preceding section, you saw how to build up an image from scratch. However, often you want to access image data for a different reason: You already have an image and you want to improve it in some way.

Of course, you can use the getPixel/getDataElements methods that you saw in the preceding section to read the image data, manipulate them, and write them back. Fortunately, the Java 2D API already supplies a number of filters that carry out common image processing operations for you.

The image manipulations all implement the BufferedImageOp interface. After you construct the operation, simply call the filter method to transform an image into another.

```
BufferedImageOp op = . . .;
BufferedImage filteredImage
   = new BufferedImage(image.getWidth(), image.getHeight(), image.
op.filter(image, filteredImage);
```

Some operations can transform an image in place (op.filter(image, image)), but most can't.

Five classes implement the BufferedImageOp interface:

```
AffineTransformOp
RescaleOp
LookupOp
ColorConvertOp
ConvolveOp
```

The AffineTransformOp carries out an affine transformation on the pixels. For example, here is how you can rotate an image about its center:

```
AffineTransform transform = AffineTransform.getRotateInstance(Ma
   image.getWidth() / 2, image.getHeight() / 2);
var op = new AffineTransformOp(transform, interpolation);
op.filter(image, filteredImage);
```

The AffineTransformOp constructor requires an affine transform and an interpolation strategy. Interpolation is necessary to determine the target image pixels if the source pixels are transformed somewhere between target pixels. For example, if you rotate source pixels, they will generally not fall exactly onto target pixels. There are three interpolation strategies: AffineTransformOp.TYPE_BICUBIC, AffineTransformOp.TYPE_BILINEAR, and AffineTransformOp.TYPE_NEAREST_NEIGHBOR. Bicubic interpolation takes a bit longer but looks better than the other two.

The program in Listing 12.22 lets you rotate an image by 5 degrees (see Figure 12.57).

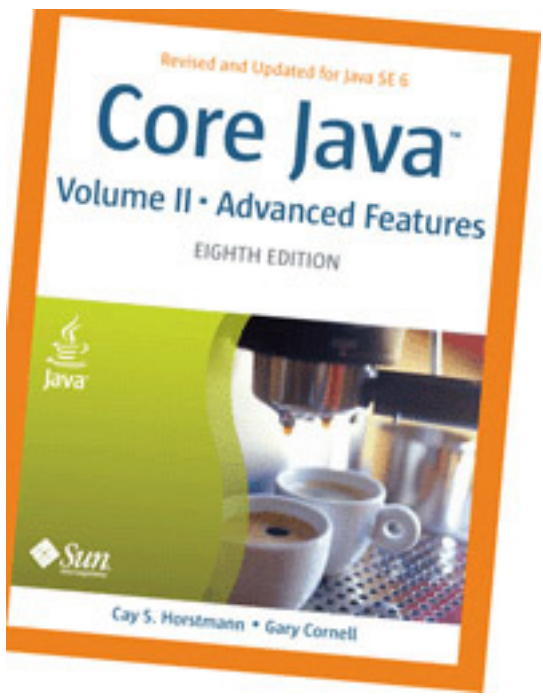


Figure 12.57: A rotated image

The RescaleOp carries out a rescaling operation

$$x_{new} = a \cdot x + b$$

for each of the color components in the image. (Alpha components are not affected.) The effect of rescaling with a > 1 is to brighten the image. Construct the RescaleOp by specifying the scaling parameters and optional rendering hints. In Listing 12.22, we use:

```
float a = 1.1f;
float b = 20.0f;
var op = new RescaleOp(a, b, null);
```

You can also supply separate scaling values for each color component—see the API notes.

The LookupOp operation lets you specify an arbitrary mapping of sample values. Supply a table that specifies how each value should be mapped. In the example program, we compute the negative of all colors, changing the color c to 255 – c.

The LookupOp constructor requires an object of type LookupTable and a map of optional hints. The LookupTable class is abstract, with two concrete subclasses: ByteLookupTable and ShortLookupTable. Since RGB color values are bytes, a ByteLookupTable should suffice. However, because of the bug described in https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6183251, we will use a ShortLookupTable instead. Here is how we construct the LookupOp for the example program:

```
var negative = new short[256];
for (int i = 0; i < 256; i++) negative[i] = (short) (255 - i);
var table = new ShortLookupTable(0, negative);
var op = new LookupOp(table, null);
```

The lookup is applied to each color component separately, but not to the alpha component. You can also supply different lookup tables for each color component—see the API notes.

You cannot apply a LookupOp to an image with an

---

indexed color model. (In those images, each sample value is an offset into a color palette.)

The ColorConvertOp is useful for color space conversions. We do not discuss it here.

The most powerful of the transformations is the ConvolveOp, which carries out a mathematical convolution. Without getting too deeply into the mathematical details, the basic idea is simple. Consider, for example, the blur filter (see Figure 12.58).

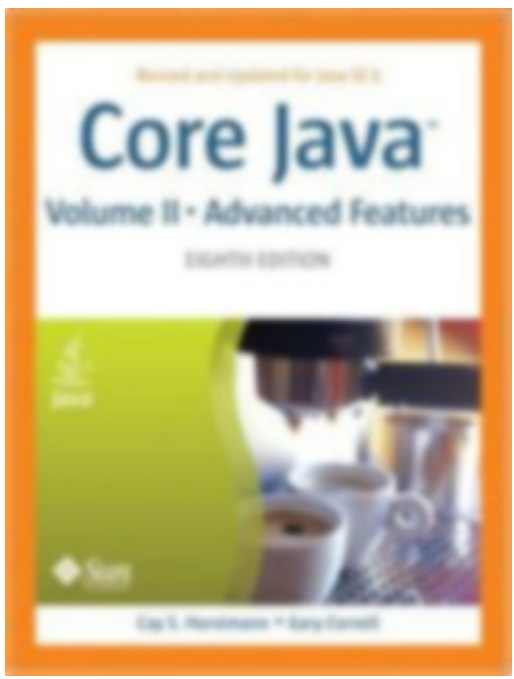


Figure 12.58: Blurring an image

The blurring is achieved by replacing each pixel with the average value from the pixel and its eight neighbors. Intuitively, it makes sense why this operation would blur out the picture. Mathematically, the averaging can be expressed as a convolution operation with the following kernel:

```
[ 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 9 1 ]
```

The kernel of a convolution is a matrix that tells what weights should be applied to the neighboring values. The kernel above produces a blurred image. A different kernel carries out edge detection, locating the areas of color changes:

[0 - 1 0 - 1 4 - 1 0 - 1 0]

Edge detection is an important technique for analyzing photographic images (see Figure 12.59).



Figure 12.59: Edge detection and inversion

To construct a convolution operation, first set up an array of the values for the kernel and construct a Kernel object. Then, construct a ConvolveOp object from the kernel and use it for filtering.

```
float[] elements = {
   0.0f, -1.0f, 0.0f,
   -1.0f,  4.f, -1.0f,
```

```
        0.0f, -1.0f, 0.0f
    };
    var kernel = new Kernel(3, 3, elements);
    var op = new ConvolveOp(kernel);
    op.filter(image, filteredImage);
```

The program in Listing 12.22 allows a user to load in a GIF or JPEG image and carry out the image manipulations that we discussed. Thanks to the power of the operations provided by Java 2D API, the program is very simple.

Listing 12.22 **v2ch12/imageProcessing/ImageProcessingFrame.java**

```
package v2ch12.imageProcessing;

import module java.base;
import module java.desktop;

/**
 * This frame has a menu to load an image and to specify various transformati
 * component to show the resulting image.
 */
public class ImageProcessingFrame extends JFrame {
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;

    private BufferedImage image;

    public ImageProcessingFrame() {
        setTitle("ImageProcessingTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        add(new JComponent() {
            public void paintComponent(Graphics g) {
                if (image != null) g.drawImage(image, 0, 0, null);
            }
        });

        var fileMenu = new JMenu("File");
        var openItem = new JMenuItem("Open");
```

```
openItem.addActionListener(_ -> openFile());
fileMenu.add(openItem);

var exitItem = new JMenuItem("Exit");
exitItem.addActionListener(_ -> System.exit(0));
fileMenu.add(exitItem);

var editMenu = new JMenu("Edit");
var blurItem = new JMenuItem("Blur");
blurItem.addActionListener(_ -> {
   float weight = 1.0f / 9.0f;
   float[] elements = new float[9];
   for (int i = 0; i < 9; i++)
      elements[i] = weight;
   convolve(elements);
});
editMenu.add(blurItem);

var sharpenItem = new JMenuItem("Sharpen");
sharpenItem.addActionListener(_ -> {
   float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 5.f, -1.0f, 0.0f, -1.0f, 0.0f };
   convolve(elements);
});
editMenu.add(sharpenItem);

var brightenItem = new JMenuItem("Brighten");
brightenItem.addActionListener(_ -> {
   float a = 1.1f;
   float b = 20.0f;
   var op = new RescaleOp(a, b, null);
   filter(op);
});
editMenu.add(brightenItem);

var edgeDetectItem = new JMenuItem("Edge detect");
edgeDetectItem.addActionListener(_ -> {
   float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 4.f, -1.0f, 0.0f, -1.0f, 0.0f };
   convolve(elements);
});
editMenu.add(edgeDetectItem);

var negativeItem = new JMenuItem("Negative");
negativeItem.addActionListener(_ -> {
   short[] negative = new short[256 * 1];
```

```
        for (int i = 0; i < 256; i++)
            negative[i] = (short) (255 - i);
        var table = new ShortLookupTable(0, negative);
        var op = new LookupOp(table, null);
        filter(op);
    });
    editMenu.add(negativeItem);

    var rotateItem = new JMenuItem("Rotate");
    rotateItem.addActionListener(_ -> {
        if (image == null) return;
        var transform = AffineTransform.getRotateInstance(Math.toRadians(5),
                image.getWidth() / 2, image.getHeight() / 2);
        var op = new AffineTransformOp(transform, AffineTransformOp.TYPE_
        filter(op);
    });
    editMenu.add(rotateItem);

    var menuBar = new JMenuBar();
    menuBar.add(fileMenu);
    menuBar.add(editMenu);
    setJMenuBar(menuBar);
}

/**
 * Open a file and load the image.
 */
public void openFile() {
    var chooser = new JFileChooser("images");
    chooser.setCurrentDirectory(new File(getClass().getPackage().getName()));
    String[] extensions = ImageIO.getReaderFileSuffixes();
    chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensio
    int r = chooser.showOpenDialog(this);
    if (r != JFileChooser.APPROVE_OPTION) return;

    try {
        Image img = ImageIO.read(chooser.getSelectedFile());
        image = new BufferedImage(img.getWidth(null), img.getHeight(null),
                BufferedImage.TYPE_INT_RGB);
        image.getGraphics().drawImage(img, 0, 0, null);
    }
    catch (IOException e) {
        JOptionPane.showMessageDialog(this, e);
    }
```

```
      repaint();
   }

   /**
    * Apply a filter and repaint.
    * @param op the image operation to apply
    */
   private void filter(BufferedImageOp op) {
      if (image == null) return;
      image = op.filter(image, null);
      repaint();
   }

   /**
    * Apply a convolution and repaint.
    * @param elements the convolution kernel (an array of 9 matrix elements)
    */
   private void convolve(float[] elements) {
      var kernel = new Kernel(3, 3, elements);
      var op = new ConvolveOp(kernel);
      filter(op);
   }
}
```

java.awt.image.BufferedImageOp 1.2

- BufferedImage filter(BufferedImage source, BufferedImage dest)
  applies the image operation to the source image and stores the result in the destination image. If dest is null, a new destination image is created. The destination image is returned.

java.awt.image.AffineTransformOp 1.2

- AffineTransformOp(AffineTransform t, int interpolationType)
  constructs an affine transform operator. The interpolation type is one of TYPE_BILINEAR, TYPE_BICUBIC, or TYPE_NEAREST_NEIGHBOR.

java.awt.image.RescaleOp 1.2

- RescaleOp(float a, float b, RenderingHints hints)
- RescaleOp(float[] as, float[] bs, RenderingHints hints)
  construct a rescale operator that carries out the scaling operation $x_{new} = a \cdot x + b$. When using the first constructor, all color components (but not the alpha component) are scaled with the same coefficients. When using the second constructor, you supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

java.awt.image.LookupOp 1.2

- LookupOp(LookupTable table, RenderingHints hints)
  constructs a lookup operator for the given lookup table.

java.awt.image.ByteLookupTable 1.2

- ByteLookupTable(int offset, byte[] data)
- ByteLookupTable(int offset, byte[][] data)
  construct a lookup table for converting byte values. The offset is subtracted from the input before the lookup. The values in the first constructor are applied to all color components but not the alpha component. When using the second constructor, supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

java.awt.image.ShortLookupTable 1.2

- ShortLookupTable(int offset, short[] data)
- ShortLookupTable(int offset, short[][] data)
  construct a lookup table for converting short values. The

offset is subtracted from the input before the lookup. The values in the first constructor are applied to all color components but not the alpha component. When using the second constructor, supply either the values for each color component, in which case the alpha component is unaffected, or the values for both alpha and color components.

java.awt.image.ConvolveOp 1.2

- ConvolveOp(Kernel kernel)
- ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)
  construct a convolution operator. The edge condition specified is one of EDGE_NO_OP and EDGE_ZERO_FILL. Edge values need to be treated specially because they don't have sufficient neighboring values to compute the convolution. The default is EDGE_ZERO_FILL.

java.awt.image.Kernel 1.2

- Kernel(int width, int height, float[] matrixElements)
  constructs a kernel for the given matrix.

12.7. Printing

In the following sections, I'll show you how you can easily print a drawing on a single sheet of paper, how you can manage a multipage printout, and how you can save a printout as a PostScript file.

12.7.1. Graphics Printing

In this section, we will tackle what is probably the most common printing situation: printing a 2D graphic. Of course,

the graphic can contain text in various fonts or even consist entirely of text.

To generate a printout, you have to take care of these two tasks:

- Supply an object that implements the Printable interface
- Start a print job

The Printable interface has a single method:

```
int print(Graphics g, PageFormat format, int page)
```

That method is called whenever the print engine needs to have a page formatted for printing. Your code draws the text and the images to be printed onto the graphics context. The page format tells you the paper size and the print margins. The page number tells you which page to render.

To start a print job, use the PrinterJob class. First, call the static getPrinterJob method to get a print job object. Then set the Printable object that you want to print.

```
Printable canvas = . . .;
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(canvas);
```

> The class PrintJob handles JDK 1.1-style printing. That class is now obsolete. Do not confuse it with the PrinterJob class.

Before starting the print job, you should call the printDialog method to display a print dialog box (see Figure 12.60). That dialog box gives the user a chance to select the printer to be used (in case multiple printers are available), the page range that should be printed, and various printer settings.

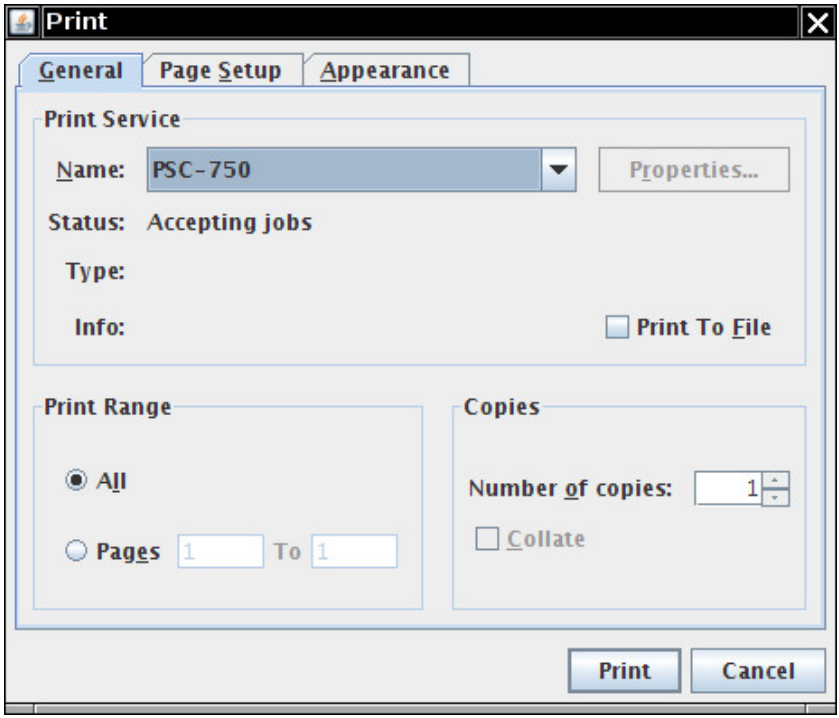


Figure 12.60: A cross-platform print dialog box

Collect printer settings in an object of a class that implements the PrintRequestAttributeSet interface, such as the HashPrintRequestAttributeSet class.

```
var attributes = new HashPrintRequestAttributeSet();
```

Add attribute settings and pass the attributes object to the printDialog method.

The printDialog method returns true if the user clicked OK and false if the user canceled the dialog box. If the user accepted, call the print method of the PrinterJob class to start the printing process. The print method might throw a PrinterException. Here is the outline of the printing code:

```
if (job.printDialog(attributes)) {
   try {
```

```
            job.print(attributes);
        }
        catch (PrinterException e) {
            . . .
        }
    }
```

Prior to JDK 1.4, the printing system used the native print and page setup dialog boxes of the host platform. To show a native print dialog box, call the printDialog method with no parameters. (There is no way to collect user settings in an attribute set.)

During printing, the print method of the PrinterJob class makes repeated calls to the print method of the Printable object associated with the job.

Since the job does not know how many pages you want to print, it simply keeps calling the print method. As long as the print method returns the value Printable.PAGE_EXISTS, the print job keeps producing pages. When the print method returns Printable.NO_SUCH_PAGE, the print job stops.

The page numbers that the print job passes to the print method start with page 0.

Therefore, the print job doesn't have an accurate page count until after the printout is complete. For that reason, the print dialog box can't display the correct page range—instead, it displays "Pages 1 to 1." You will see in the next section how to avoid this blemish by supplying a Book object to the print job.

During the printing process, the print job repeatedly calls the print method of the Printable object. The print job is allowed to make multiple calls for the same page. You should therefore not count pages inside the print method but always rely on the page number parameter. There is a good reason why the print job might call the print method repeatedly for the same page. Some printers, in particular dot-matrix and inkjet printers, use

banding. They print one band at a time, advance the paper, and then print the next band. The print job might use banding even for laser printers that print a full page at a time—it gives the print job a way of managing the size of the spool file.

If the print job needs the Printable object to print a band, it sets the clip area of the graphics context to the requested band and calls the print method. Its drawing operations are clipped against the band rectangle, and only those drawing elements that show up in the band are rendered. Your print method need not be aware of that process, with one caveat: It should not interfere with the clip area.

> The Graphics object that your print method gets is also clipped against the page margins. If you replace the clip area, you can draw outside the margins. Especially in a printer graphics context, the clipping area must be respected. Call clip, not setClip, to further restrict the clipping area. If you must remove a clip area, make sure to call getClip at the beginning of your print method and restore that clip area.

The PageFormat parameter of the print method contains information about the printed page. The methods getWidth and getHeight return the paper size, measured in points. (One point is 1/72 of an inch; an inch equals 25.4 millimeters.) For example, A4 paper is approximately 595 × 842 points, and US Letter paper is 612 × 792 points.

Points are a common measurement in the printing trade in the United States. Much to the chagrin of the rest of the world, the printing package uses point units. There are two purposes for that: paper sizes and paper margins are measured in points, and points are the default unit for all print graphics contexts. You can verify that in the example program at the end of this section. The program prints two lines of text that are 72 units apart. Run the example program and measure the distance between the baselines; they are exactly 1 inch or 25.4 millimeters apart.

The getWidth and getHeight methods of the PageFormat class give you the complete paper size. Not all of the paper area is printable. Users typically select margins, and even if they don't, printers need to somehow grip the sheets of paper on which they print and therefore have a small unprintable area around the edges.

The methods getImageableWidth and getImageableHeight tell you the dimensions of the area that you can actually fill. However, the margins need not be symmetrical, so you must also know the top left corner of the imageable area (see Figure 12.61), which you obtain by the methods getImageableX and getImageableY.

.

Figure 12.61: Page format measurements

The graphics context that you receive in the print method is clipped to exclude the margins, but the origin of the coordinate system is nevertheless the top left corner of the paper. It makes sense to translate the coordinate system to the top left corner of the imageable area. Simply start your print method with

```
g.translate(pageFormat.getImageableX(), pageFormat.getImag
```

If you want your users to choose the settings for the page margins or to switch between portrait and landscape orientation without setting other printing attributes, call the pageDialog method of the PrinterJob class:

```
PageFormat format = job.pageDialog(attributes);
```

One of the tabs of the print dialog box contains the page setup dialog (see Figure 12.62). You might still want to give users an option to set the page format before printing, especially if your program presents a "what you

see is what you get" display of the pages to be printed. The pageDialog method returns a PageFormat object with the user settings.

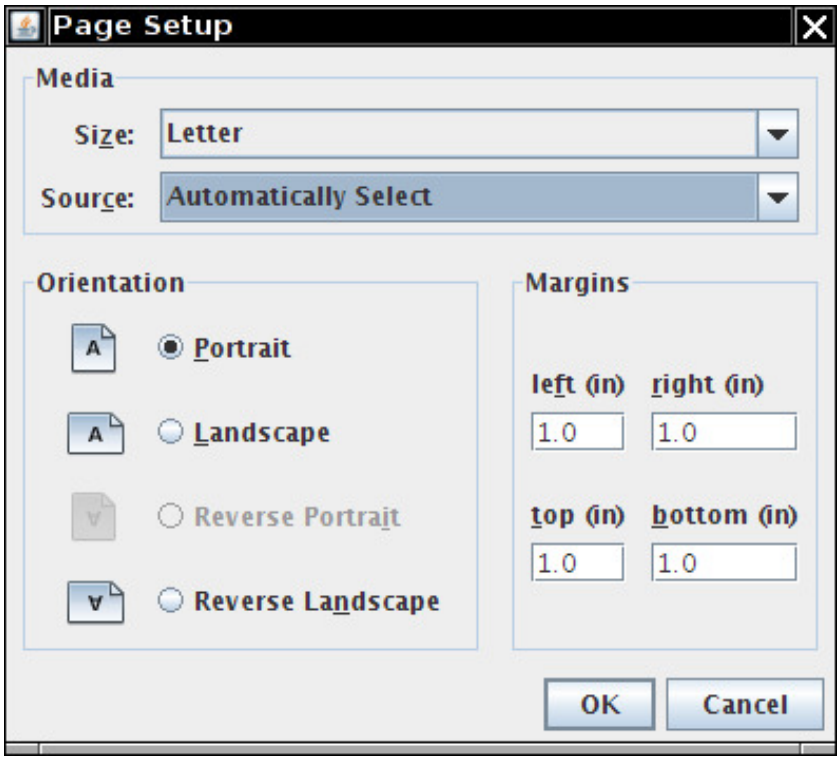


Figure 12.62: A cross-platform page setup dialog

The program at the end of this section shows how to render the same set of shapes on the screen and on the printed page. A subclass of JPanel implements the Printable interface. Both the paintComponent and the print methods call the same method to carry out the actual drawing.

```
class PrintPanel extends JPanel implements Printable {
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      var g2 = (Graphics2D) g;
      drawPage(g2);
```

```
    }

    public int print(Graphics g, PageFormat pf, int page) throws Printe
       if (page >= 1) return Printable.NO_SUCH_PAGE;
       var g2 = (Graphics2D) g;
       g2.translate(pf.getImageableX(), pf.getImageableY());
       drawPage(g2);
       return Printable.PAGE_EXISTS;
    }

    public void drawPage(Graphics2D g2) {
       // shared drawing code goes here
       . . .
    }
    . . .
}
```

This example displays and prints the image shown in Figure 12.50—namely, the outline of the message "Hello, World" used as a clipping area for a pattern of lines.

Click the Print button to start printing, or click the Page setup button to open the page setup dialog box. Listing 12.23 shows the code.

To show a native page setup dialog box, pass a default PageFormat object to the pageDialog method. The method clones that object, modifies it according to the user selections in the dialog box, and returns the cloned object.

```
PageFormat defaultFormat = printJob.defaultPage();
PageFormat selectedFormat = printJob.pageDialog(defaultFor
```

Listing 12.23 **v2ch12/print/PrintDemoFrame.java**

```
package v2ch12.print;
```

```
import module java.desktop;

/**
 * This frame shows a panel with 2D graphics and buttons to print the graphics
 * the page format.
 */
public class PrintDemoFrame extends JFrame {
   private PrintComponent canvas;
   private PrintRequestAttributeSet attributes;

   public PrintDemoFrame() {
      canvas = new PrintComponent();
      add(canvas, BorderLayout.CENTER);

      attributes = new HashPrintRequestAttributeSet();

      var buttonPanel = new JPanel();
      var printButton = new JButton("Print");
      buttonPanel.add(printButton);
      printButton.addActionListener(_ -> {
         try {
            PrinterJob job = PrinterJob.getPrinterJob();
            job.setPrintable(canvas);
            if (job.printDialog(attributes)) job.print(attributes);
         }
         catch (PrinterException e) {
            JOptionPane.showMessageDialog(PrintDemoFrame.this, e);
         }
      });

      var pageSetupButton = new JButton("Page setup");
      buttonPanel.add(pageSetupButton);
      pageSetupButton.addActionListener(_ -> {
         PrinterJob job = PrinterJob.getPrinterJob();
         job.pageDialog(attributes);
      });

      add(buttonPanel, BorderLayout.NORTH);
      pack();
   }
}
```

Listing 12.24 **v2ch12/print/PrintComponent.java**

```
package v2ch12.print;

import module java.desktop;

/**
 * This component generates a 2D graphics image for screen display and printi
 */
public class PrintComponent extends JComponent implements Printable {
   private static final Dimension PREFERRED_SIZE = new Dimension(300, 300);

   public void paintComponent(Graphics g) {
      var g2 = (Graphics2D) g;
      drawPage(g2);
   }

   public int print(Graphics g, PageFormat pf, int page) throws PrinterExceptio
      if (page >= 1) return Printable.NO_SUCH_PAGE;
      var g2 = (Graphics2D) g;
      g2.translate(pf.getImageableX(), pf.getImageableY());
      g2.draw(new Rectangle2D.Double(0, 0, pf.getImageableWidth(), pf.getImag

      drawPage(g2);
      return Printable.PAGE_EXISTS;
   }

   /**
    * This method draws the page both on the screen and the printer graphics c
    * @param g2 the graphics context
    */
   public void drawPage(Graphics2D g2) {
      FontRenderContext context = g2.getFontRenderContext();
      var f = new Font("Serif", Font.PLAIN, 72);
      var clipShape = new GeneralPath();

      var layout = new TextLayout("Hello", f, context);
      AffineTransform transform = AffineTransform.getTranslateInstance(0, 72)
      Shape outline = layout.getOutline(transform);
      clipShape.append(outline, false);

      layout = new TextLayout("World", f, context);
      transform = AffineTransform.getTranslateInstance(0, 144);
      outline = layout.getOutline(transform);
      clipShape.append(outline, false);
```

```
      g2.draw(clipShape);
      g2.clip(clipShape);

      final int NLINES = 50;
      var p = new Point2D.Double(0, 0);
      for (int i = 0; i < NLINES; i++) {
         double x = (2 * getWidth() * i) / NLINES;
         double y = (2 * getHeight() * (NLINES - 1 - i)) / NLINES;
         var q = new Point2D.Double(x, y);
         g2.draw(new Line2D.Double(p, q));
      }
   }

   public Dimension getPreferredSize() {
      return PREFERRED_SIZE;
   }
}
```

java.awt.print.Printable 1.2

- int print(Graphics g, PageFormat format, int pageNumber)
  renders a page and returns PAGE_EXISTS, or returns NO_SUCH_PAGE.

java.awt.print.PrinterJob 1.2

- static PrinterJob getPrinterJob()
  returns a printer job object.
- PageFormat defaultPage()
  returns the default page format for this printer.
- boolean printDialog(PrintRequestAttributeSet attributes)
- boolean printDialog()
  open a print dialog box to allow a user to select the pages to be printed and to change print settings. The first method displays a cross-platform dialog box, the second a native dialog box. The first method modifies the attributes object to reflect the user settings. Both methods return true if the user accepts the dialog box.
- PageFormat pageDialog(PrintRequestAttributeSet attributes)

- PageFormat pageDialog(PageFormat defaults)
  display a page setup dialog box. The first method displays a cross-platform dialog box, the second a native dialog box. Both methods return a PageFormat object with the format that the user requested in the dialog box. The first method modifies the attributes object to reflect the user settings. The second method does not modify the defaults object.
- void setPrintable(Printable p)
- void setPrintable(Printable p, PageFormat format)
  set the Printable of this print job and an optional page format.
- void print()
- void print(PrintRequestAttributeSet attributes)
  print the current Printable by repeatedly calling its print method and sending the rendered pages to the printer, until no more pages are available.

java.awt.print.PageFormat **1.2**

- double getWidth()
- double getHeight()
  return the width and height of the page.
- double getImageableWidth()
- double getImageableHeight()
  return the width and height of the imageable area of the page.
- double getImageableX()
- double getImageableY()
  return the position of the top left corner of the imageable area.
- int getOrientation()
  returns one of PORTRAIT, LANDSCAPE, or REVERSE_LANDSCAPE. Page orientation is transparent to programmers because the page format and graphics context settings automatically reflect the page orientation.

12.7.2. Multiple-Page Printing

In practice, you usually don't pass a raw Printable object to a print job. Instead, you should obtain an object of a class that implements the Pageable interface. The Java platform supplies one such class, called Book. A book is made up of sections, each of which is a Printable object. To make a book, add Printable objects and their page counts.

```
var book = new Book();
Printable coverPage = . . .;
Printable bodyPages = . . .;
book.append(coverPage, pageFormat); // append 1 page
book.append(bodyPages, pageFormat, pageCount);
```

Then, use the setPageable method to pass the Book object to the print job.

```
printJob.setPageable(book);
```

Now the print job knows exactly how many pages to print, so the print dialog box displays an accurate page range and the user can select the entire range or subranges.

> When the print job calls the print methods of the Printable sections, it passes the current page number of the book, and not of each section, as the current page number. That is a huge pain—each section must know the page counts of the preceding sections to make sense of the page number parameter.

From your perspective as a programmer, the biggest challenge of using the Book class is that you must know how many pages each section will have when you print it. Your Printable class needs a layout algorithm that computes the layout of the material on the printed pages. Before printing starts, invoke that algorithm to compute the page breaks and the page count. You can retain the layout information so you have it handy during the printing process.

You must guard against the possibility that the user has changed the page format. If that happens, you must recompute the layout, even if the information that you want to print has not changed.

Listing 12.26 shows how to produce a multipage printout. This program prints a message in very large characters on a number of pages (see Figure 12.63). You can then trim the margins and tape the pages together to form a banner.

.

Figure 12.63: A banner

The layoutPages method of the Banner class computes the layout. We first lay out the message string in a 72-point font. We then compute the height of the resulting string and compare it with the imageable height of the page. We derive a scale factor from these two measurements. When printing the string, we magnify it by that scale factor.

To lay out your information precisely, you usually need access to the printer graphics context. Unfortunately, there is no way to obtain that graphics context before printing actually starts. In our example program, we make do with the screen graphics context and hope that the font metrics of the screen and printer match.

The getPageCount method of the Banner class first calls the layout method. Then it scales up the width of the string and divides it by the imageable width of each page. The quotient, rounded up to the next integer, is the page count.

It sounds like it might be difficult to print the banner because characters can be broken across multiple pages. However, thanks to the power of the Java 2D API, this turns out not to be a problem at all. When a particular page is requested, we simply use the translate method of the Graphics2D class to shift the top left corner of the string to the left. Then, we set a clip

rectangle that equals the current page (see Figure 12.64). Finally, we scale the graphics context with the scale factor that the layout method computed.

.

Figure 12.64: Printing a page of a banner

This example shows the power of transformations. The drawing code is kept simple, and the transformation does all the work of placing the drawing in the appropriate place. Finally, the clip cuts away the part of the image that falls outside the page. This program shows another compelling use of transformations—to display a print preview.

Listing 12.25 **v2ch12/book/BookDemoFrame.java**

```
package v2ch12.book;

import module java.desktop;

/**
 * This frame has a text field for the banner text and buttons for printing, page
 * print preview.
 */
public class BookDemoFrame extends JFrame {
   private JTextField text;
   private PageFormat pageFormat;
   private PrintRequestAttributeSet attributes;

   public BookDemoFrame() {
      text = new JTextField();
      add(text, BorderLayout.NORTH);

      attributes = new HashPrintRequestAttributeSet();

      var buttonPanel = new JPanel();

      var printButton = new JButton("Print");
      buttonPanel.add(printButton);
```

```
      printButton.addActionListener(_ -> {
         try {
            PrinterJob job = PrinterJob.getPrinterJob();
            job.setPageable(makeBook());
            if (job.printDialog(attributes)) {
               job.print(attributes);
            }
         }
         catch (PrinterException e) {
            JOptionPane.showMessageDialog(BookDemoFrame.this, e);
         }
      });

      var pageSetupButton = new JButton("Page setup");
      buttonPanel.add(pageSetupButton);
      pageSetupButton.addActionListener(_ -> {
         PrinterJob job = PrinterJob.getPrinterJob();
         pageFormat = job.pageDialog(attributes);
      });

      var printPreviewButton = new JButton("Print preview");
      buttonPanel.add(printPreviewButton);
      printPreviewButton.addActionListener(_ -> {
         var dialog = new PrintPreviewDialog(makeBook());
         dialog.setVisible(true);
      });

      add(buttonPanel, BorderLayout.SOUTH);
      pack();
   }

   /**
    * Makes a book that contains a cover page and the pages for the banner.
    */
   public Book makeBook() {
      if (pageFormat == null) {
         PrinterJob job = PrinterJob.getPrinterJob();
         pageFormat = job.defaultPage();
      }
      var book = new Book();
      String message = text.getText();
      var banner = new Banner(message);
      int pageCount = banner.getPageCount((Graphics2D) getGraphics(), pageFo
      book.append(new CoverPage(message + " (" + pageCount + " pages)"), page
```

```
      book.append(banner, pageFormat, pageCount);
      return book;
   }
}
```

Listing 12.26 **v2ch12/book/Banner.java**

```
package v2ch12.book;

import module java.desktop;

/**
 * A banner that prints a text string on multiple pages.
 */
public class Banner implements Printable {
   private String message;
   private double scale;

   /**
    * Constructs a banner.
    * @param m the message string
    */
   public Banner(String m) {
      message = m;
   }

   /**
    * Gets the page count of this section.
    * @param g2 the graphics context
    * @param pf the page format
    * @return the number of pages needed
    */
   public int getPageCount(Graphics2D g2, PageFormat pf) {
      if (message.equals("")) return 0;
      FontRenderContext context = g2.getFontRenderContext();
      var f = new Font("Serif", Font.PLAIN, 72);
      Rectangle2D bounds = f.getStringBounds(message, context);
      scale = pf.getImageableHeight() / bounds.getHeight();
      double width = scale * bounds.getWidth();
      int pages = (int) Math.ceil(width / pf.getImageableWidth());
      return pages;
   }
```

```
public int print(Graphics g, PageFormat pf, int page) throws PrinterExceptio
   var g2 = (Graphics2D) g;
   if (page > getPageCount(g2, pf)) return Printable.NO_SUCH_PAGE;
   g2.translate(pf.getImageableX(), pf.getImageableY());

   drawPage(g2, pf, page);
   return Printable.PAGE_EXISTS;
}

public void drawPage(Graphics2D g2, PageFormat pf, int page) {
   if (message.equals("")) return;
   page--; // account for cover page

   drawCropMarks(g2, pf);
   g2.clip(new Rectangle2D.Double(0, 0, pf.getImageableWidth(), pf.getImage
   g2.translate(-page * pf.getImageableWidth(), 0);
   g2.scale(scale, scale);
   FontRenderContext context = g2.getFontRenderContext();
   var f = new Font("Serif", Font.PLAIN, 72);
   var layout = new TextLayout(message, f, context);
   AffineTransform transform
      = AffineTransform.getTranslateInstance(0, layout.getAscent());
   Shape outline = layout.getOutline(transform);
   g2.draw(outline);
}

/**
 * Draws 1/2" crop marks in the corners of the page.
 * @param g2 the graphics context
 * @param pf the page format
 */
public void drawCropMarks(Graphics2D g2, PageFormat pf) {
   final double C = 36; // crop mark length = 1/2 inch
   double w = pf.getImageableWidth();
   double h = pf.getImageableHeight();
   g2.draw(new Line2D.Double(0, 0, 0, C));
   g2.draw(new Line2D.Double(0, 0, C, 0));
   g2.draw(new Line2D.Double(w, 0, w, C));
   g2.draw(new Line2D.Double(w, 0, w - C, 0));
   g2.draw(new Line2D.Double(0, h, 0, h - C));
   g2.draw(new Line2D.Double(0, h, C, h));
   g2.draw(new Line2D.Double(w, h, w, h - C));
   g2.draw(new Line2D.Double(w, h, w - C, h));
}
```

```
}

/**
 * This class prints a cover page with a title.
 */
class CoverPage implements Printable {
   private String title;

   /**
    * Constructs a cover page.
    * @param t the title
    */
   public CoverPage(String t) {
      title = t;
   }

   public int print(Graphics g, PageFormat pf, int page) throws PrinterExceptio
      if (page >= 1) return Printable.NO_SUCH_PAGE;
      var g2 = (Graphics2D) g;
      g2.setPaint(Color.black);
      g2.translate(pf.getImageableX(), pf.getImageableY());
      FontRenderContext context = g2.getFontRenderContext();
      Font f = g2.getFont();
      var layout = new TextLayout(title, f, context);
      float ascent = layout.getAscent();
      g2.drawString(title, 0, ascent);
      return Printable.PAGE_EXISTS;
   }
}
```

Listing 12.27 **v2ch12/book/PrintPreviewDialog.java**

```
package v2ch12.book;

import module java.desktop;

/**
 * This class implements a generic print preview dialog.
 */
public class PrintPreviewDialog extends JDialog {
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
```

```
private PrintPreviewCanvas canvas;

/**
 * Constructs a print preview dialog.
 * @param p a Printable
 * @param pf the page format
 * @param pages the number of pages in p
 */
public PrintPreviewDialog(Printable p, PageFormat pf, int pages) {
   var book = new Book();
   book.append(p, pf, pages);
   layoutUI(book);
}

/**
 * Constructs a print preview dialog.
 * @param b a Book
 */
public PrintPreviewDialog(Book b) {
   layoutUI(b);
}

/**
 * Lays out the UI of the dialog.
 * @param book the book to be previewed
 */
public void layoutUI(Book book) {
   setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

   canvas = new PrintPreviewCanvas(book);
   add(canvas, BorderLayout.CENTER);

   var buttonPanel = new JPanel();

   var nextButton = new JButton("Next");
   buttonPanel.add(nextButton);
   nextButton.addActionListener(_ -> canvas.flipPage(1));

   var previousButton = new JButton("Previous");
   buttonPanel.add(previousButton);
   previousButton.addActionListener(_ -> canvas.flipPage(-1));

   var closeButton = new JButton("Close");
   buttonPanel.add(closeButton);
```

```
      closeButton.addActionListener(_ -> setVisible(false));

      add(buttonPanel, BorderLayout.SOUTH);
   }
}
```

Listing 12.28 **v2ch12/book/PrintPreviewCanvas.java**

```
package v2ch12.book;

import module java.desktop;

/**
 * The canvas for displaying the print preview.
 */
public class PrintPreviewCanvas extends JComponent {
   private Book book;
   private int currentPage;

   /**
    * Constructs a print preview canvas.
    * @param b the book to be previewed
    */
   public PrintPreviewCanvas(Book b) {
      book = b;
      currentPage = 0;
   }

   public void paintComponent(Graphics g) {
      var g2 = (Graphics2D) g;
      PageFormat pageFormat = book.getPageFormat(currentPage);

      double xoff; // x offset of page start in window
      double yoff; // y offset of page start in window
      double scale; // scale factor to fit page in window
      double px = pageFormat.getWidth();
      double py = pageFormat.getHeight();
      double sx = getWidth() - 1;
      double sy = getHeight() - 1;
      if (px / py < sx / sy) // center horizontally
      {
         scale = sy / py;
         xoff = 0.5 * (sx - scale * px);
```

```
            yoff = 0;
        }
        else
        // center vertically
        {
            scale = sx / px;
            xoff = 0;
            yoff = 0.5 * (sy - scale * py);
        }
        g2.translate((float) xoff, (float) yoff);
        g2.scale((float) scale, (float) scale);

        // draw page outline (ignoring margins)
        var page = new Rectangle2D.Double(0, 0, px, py);
        g2.setPaint(Color.white);
        g2.fill(page);
        g2.setPaint(Color.black);
        g2.draw(page);

        Printable printable = book.getPrintable(currentPage);
        try {
            printable.print(g2, pageFormat, currentPage);
        }
        catch (PrinterException e) {
            g2.draw(new Line2D.Double(0, 0, px, py));
            g2.draw(new Line2D.Double(px, 0, 0, py));
        }
    }

    /**
     * Flip the book by the given number of pages.
     * @param by the number of pages to flip by. Negative values flip backwards
     */
    public void flipPage(int by) {
        int newPage = currentPage + by;
        if (0 <= newPage && newPage < book.getNumberOfPages()) {
            currentPage = newPage;
            repaint();
        }
    }
}
```

12.7.3. Print Services

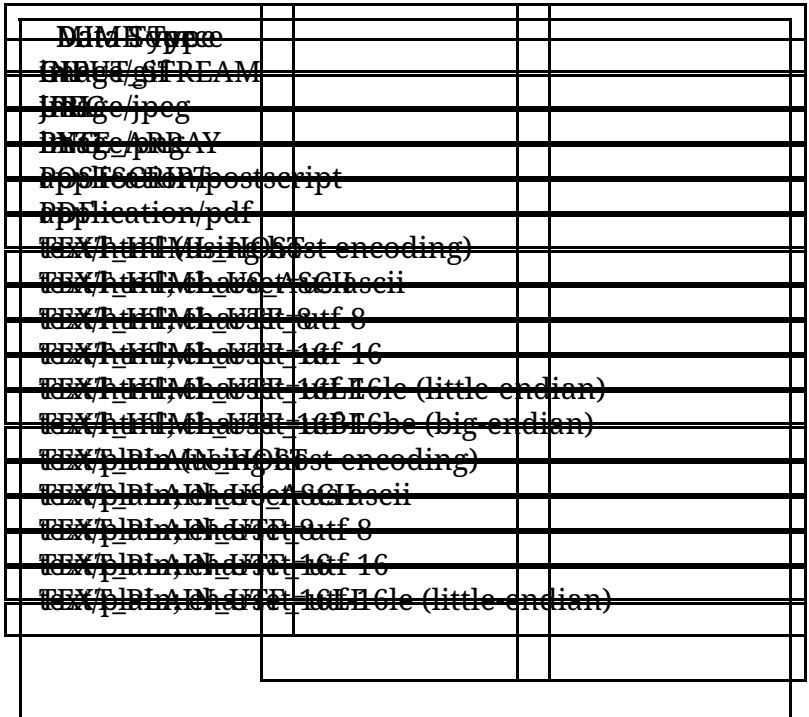
So far, you have seen how to print 2D graphics. However, the printing API affords far greater flexibility. The API defines a number of data types and lets you find print services that are able to print them. Among the data types are

- Images in GIF, JPEG, or PNG format
- Documents in text, HTML, PostScript, or PDF format
- Raw printer code data
- Objects of a class that implements Printable, Pageable, or RenderableImage

The data themselves can be stored in a source of bytes or characters such as an input stream, a URL, or an array. A document flavor describes the combination of a data source and a data type. The DocFlavor class defines a number of inner classes for the various data sources. Each of the inner classes defines constants to specify the flavors. For example, the constant

```
DocFlavor.INPUT_STREAM.GIF
```

describes a GIF image that is read from an input stream. Table 12.4 lists the combinations.



text/plain;charset=utf-16be	Text, UTF-16 big-endian
application/vnd.hp-PCL	PCL (Hewlett Packard Printer Control Language)
application/octet-stream	Raw printer data
text/html;charset=utf-16	HTML, UTF-16
text/plain;charset=utf-16	Text, UTF-16
CHAR_ARRAY	
N/A	FORMATTED
N/A	PAGEABLE
N/A	RENDERABLE_IMAGE

Table 12.4: Document Flavors for Print Services

Suppose you want to print a GIF image located in a file. First, find out whether there is a print service that is capable of handling the task. The static lookupPrintServices method of the PrintServiceLookup class returns an array of PrintService objects that can handle the given document flavor.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
PrintService[] services = PrintServiceLookup.lookupPrintServices(fla
```

The second argument of the lookupPrintServices method is null to indicate that we don't want to constrain the search by specifying printer attributes. We'll cover attributes in the next section.

If the lookup yields an array with more than one element, select from the listed print services. You can call the getName method of the PrintService class to get the printer names and let the user choose.

Next, get a document print job from the service:

```
DocPrintJob job = services[i].createPrintJob();
```

For printing, you need an object that implements the Doc interface. The Java library supplies a class SimpleDoc for that purpose. The SimpleDoc constructor requires the data source object, the document flavor, and an optional attribute set. For example,

```
var in = new FileInputStream(fileName);
var doc = new SimpleDoc(in, flavor, null);
```

Finally, you are ready to print:

```
job.print(doc, null);
```

As before, the null argument can be replaced by an attribute set.

Note that this printing process is quite different from that of the preceding section. There is no user interaction through print dialog boxes. For example, you can implement a server-side printing mechanism in which users submit print jobs through a web form.

javax.print.PrintServiceLookup1.4

- PrintService[] lookupPrintServices(DocFlavor flavor, AttributeSet attributes)
  looks up the print services that can handle the given document flavor and attributes. The attributes parameter can be null if there are no attributes to consider.

javax.print.PrintService1.4

- DocPrintJob createPrintJob()
  creates a print job for printing an object of a class that implements the Doc interface, such as a SimpleDoc.

javax.print.DocPrintJob1.4

- void print(Doc doc, PrintRequestAttributeSet attributes)
  prints the given document with the given attributes. The attributes parameter can be null if no printing attributes are required.

javax.print.SimpleDoc1.4

- SimpleDoc(Object data, DocFlavor flavor, DocAttributeSet attributes)
  constructs a SimpleDoc object that can be printed with a DocPrintJob. The data parameter is the object with the print data, such as an input stream or a Printable. The attributes parameter can be null if attributes are not required.

12.7.4. Stream Print Services

A print service sends print data to a printer. A stream print service generates the same print data but instead sends them to a stream, perhaps for delayed printing or because the print data format can be interpreted by other programs. In particular, if the print data format is PostScript, it may be useful to save the print data to a file because many programs can process PostScript files. The Java platform includes a stream print service that can produce PostScript output from images and 2D graphics. You can use that service on all systems, even if there are no local printers.

Enumerating stream print services is a bit more tedious than locating regular print services. You need both the DocFlavor of the object to be printed and the MIME type of the stream output. You then get a StreamPrintServiceFactory array of factories.

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories
   = StreamPrintServiceFactory.lookupStreamPrintServiceFactories(f
```

The StreamPrintServiceFactory class has no methods that would help us distinguish any one factory from another, so we just take factories[0]. We call the getPrintService method with an output stream parameter to get a StreamPrintService object.

```
var out = new FileOutputStream(fileName);
StreamPrintService service = factories[0].getPrintService(out);
```

The StreamPrintService class is a subclass of PrintService. To produce a printout, simply follow the steps of the preceding section.

javax.print.StreamPrintServiceFactory1.4

- StreamPrintServiceFactory[] lookupStreamPrintServiceFactories(DocFlavor flavor, String mimeType)
  looks up the stream print service factories that can print the given document flavor and produce an output stream of the given MIME type.
- StreamPrintService getPrintService(OutputStream out)
  gets a print service that sends the printing output to the given output stream.

The program in Listing 12.29 demonstrates how to use a stream print service to print Java 2D shapes to a PostScript file. You can replace the sample drawing code with code that generates any Java 2D shapes and have the shapes converted to PostScript. Then you can easily convert the result to PDF or EPS, using an external tool. (Unfortunately, Java does not support printing to PDF directly.)

In this example, we call a draw method that draws Java 2D shapes onto a Graphics2D object. If you want to draw the surface of a component (for example, a table or tree), use the following code:

```
private static int IMAGE_WIDTH = component.getWidth();
private static int IMAGE_HEIGHT = component.getHeight();
public static void draw(Graphics2D g2) { component.paint(g2)
```

Listing 12.29 **v2ch12/printService/PrintServiceDemo.java**

```
package v2ch12.printService;

import module java.base;
import module java.desktop;

/**
 * This program demonstrates the use of stream print services. The program pr
 * shapes to a PostScript file. If you don't supply a file name on the command li
 * is saved to out.ps.
 */
class PrintServiceDemo {
   // Set your image dimensions here
   int IMAGE_WIDTH = 300;
   int IMAGE_HEIGHT = 300;

   void main(String[] args) throws Exception {
      String fileName = args.length > 0 ? args[0] : "out.ps";
      DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
      String mimeType = "application/postscript";
      StreamPrintServiceFactory[] factories = StreamPrintServiceFactory
            .lookupStreamPrintServiceFactories(flavor, mimeType);
      var out = new FileOutputStream(fileName);
      if (factories.length > 0) {
         PrintService service = factories[0].getPrintService(out);
         var doc = new SimpleDoc(new Printable() {
            public int print(Graphics g, PageFormat pf, int page) {
               if (page >= 1)
                  return Printable.NO_SUCH_PAGE;
               else {
                  double sf1 = pf.getImageableWidth() / (IMAGE_WIDTH + 1);
                  double sf2 = pf.getImageableHeight() / (IMAGE_HEIGHT + 1);
                  double s = Math.min(sf1, sf2);
                  var g2 = (Graphics2D) g;
                  g2.translate((pf.getWidth() - pf.getImageableWidth()) / 2,
                        (pf.getHeight() - pf.getImageableHeight()) / 2);
                  g2.scale(s, s);

                  draw(g2);
                  return Printable.PAGE_EXISTS;
               }
            }
         }, flavor, null);
```

```
            DocPrintJob job = service.createPrintJob();
            var attributes = new HashPrintRequestAttributeSet();
            job.print(doc, attributes);
         }
         else
            IO.println("No factories for " + mimeType);
      }
      void draw(Graphics2D g2) {
         // Your drawing instructions go here
         FontRenderContext context = g2.getFontRenderContext();
         var f = new Font("Serif", Font.PLAIN, 72);
         var clipShape = new GeneralPath();

         var layout = new TextLayout("Hello", f, context);
         AffineTransform transform = AffineTransform.getTranslateInstance(0, 72)
         Shape outline = layout.getOutline(transform);
         clipShape.append(outline, false);

         layout = new TextLayout("World", f, context);
         transform = AffineTransform.getTranslateInstance(0, 144);
         outline = layout.getOutline(transform);
         clipShape.append(outline, false);

         g2.draw(clipShape);
         g2.clip(clipShape);

         final int NLINES = 50;
         var p = new Point2D.Double(0, 0);
         for (int i = 0; i < NLINES; i++) {
            double x = (2 * IMAGE_WIDTH * i) / NLINES;
            double y = (2 * IMAGE_HEIGHT * (NLINES - 1 - i)) / NLINES;
            var q = new Point2D.Double(x, y);
            g2.draw(new Line2D.Double(p, q));
         }
      }
}
```

12.7.5. Printing Attributes

The print service API contains a complex set of interfaces and classes to specify various kinds of attributes. There are four important groups of attributes. The first two specify requests to the printer.

- Print request attributes request particular features for all doc objects in a print job, such as two-sided printing or the paper size.
- Doc attributes are request attributes that apply only to a single doc object.

The other two attributes contain information about the printer and job status.

- Print service attributes give information about the print service, such as the printer make and model or whether the printer is currently accepting jobs.
- Print job attributes give information about the status of a particular print job, such as whether the job is already completed.

To describe the various attributes, there is an interface Attribute with subinterfaces:

```
PrintRequestAttribute
DocAttribute
PrintServiceAttribute
PrintJobAttribute
SupportedValuesAttribute
```

Individual attribute classes implement one or more of these interfaces. For example, objects of the Copies class describe the number of copies of a printout. That class implements both the PrintRequestAttribute and the PrintJobAttribute interfaces. Clearly, a print request can contain a request for multiple copies. Conversely, an attribute of the print job might be how many of these copies were actually printed. That number might be lower, perhaps because of printer limitations or because the printer ran out of paper.

The SupportedValuesAttribute interface indicates that an attribute value does not reflect actual request or status data but rather the capability of a service. For example, the CopiesSupported class implements the SupportedValuesAttribute interface. An object of that class might describe that a printer supports 1 through 99 copies of a

printout.

Figure 12.65 shows a class diagram of the attribute hierarchy.

.

Figure 12.65: The attribute hierarchy

In addition to the interfaces and classes for individual attributes, the print service API defines interfaces and classes for attribute sets. A superinterface, AttributeSet, has four subinterfaces:

PrintRequestAttributeSet
DocAttributeSet
PrintServiceAttributeSet
PrintJobAttributeSet

Each of these interfaces has an implementing class, yielding the five classes:

HashAttributeSet
HashPrintRequestAttributeSet
HashDocAttributeSet
HashPrintServiceAttributeSet
HashPrintJobAttributeSet

Figure 12.66 shows a class diagram of the attribute set hierarchy.

.

Figure 12.66: The attribute set hierarchy

For example, you can construct a print request attribute set like this:

```
var attributes = new HashPrintRequestAttributeSet();
```

After constructing the set, you are freed from worrying about

the Hash prefix.

Why have all these interfaces? They make it possible to check for correct attribute usage. For example, a DocAttributeSet accepts only objects that implement the DocAttribute interface. Any attempt to add another attribute results in a runtime error.

An attribute set is a specialized kind of map where the keys are of type Class and the values belong to a class that implements the Attribute interface. For example, if you insert an object

```
new Copies(10)
```

into an attribute set, then its key is the Class object Copies.class. That key is called the category of the attribute. The Attribute interface declares a method

```
Class getCategory()
```

that returns the category of an attribute. The Copies class defines the method to return the object Copies.class, but it isn't a requirement that the category be the same as the class of the attribute.

When an attribute is added to an attribute set, the category is extracted automatically. Just add the attribute value:

```
attributes.add(new Copies(10));
```

If you subsequently add another attribute with the same category, it overwrites the first one.

To retrieve an attribute, you need to use the category as the key, for example:

```
AttributeSet attributes = job.getAttributes();
var copies = (Copies) attributes.get(Copies.class);
```

Finally, attributes are organized by the values they can have. The Copies attribute can have any integer value. The Copies class extends the IntegerSyntax class that takes care of all integer-valued attributes. The getValue method returns the

integer value of the attribute, for example:

```
int n = copies.getValue();
```

The classes

```
TextSyntax
DateTimeSyntax
URISyntax
```

encapsulate a string, a date and time value, or a URI.

Finally, many attributes can take a finite number of values. For example, the PrintQuality attribute has three settings: draft, normal, and high. They are represented by three constants:

```
PrintQuality.DRAFT
PrintQuality.NORMAL
PrintQuality.HIGH
```

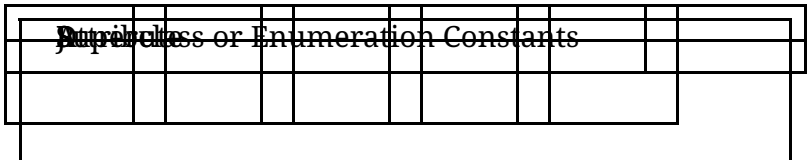
Attribute classes with a finite number of values extend the EnumSyntax class, which provides a number of convenience methods to set up these enumerations in a typesafe manner. You need not worry about the mechanism when using such an attribute. Simply add the named values to attribute sets:

```
attributes.add(PrintQuality.HIGH);
```

Here is how you check the value of an attribute:

```
if (attributes.get(PrintQuality.class) == PrintQuality.HIGH)
   . . .
```

Table 12.5 lists the printing attributes. The second column lists the superclass of the attribute class (for example, IntegerSyntax for the Copies attribute) or the set of enumeration values for the attributes with a finite set of values. The last four columns indicate whether the attribute class implements the DocAttribute (D), PrintJobAttribute (J), PrintRequestAttribute (R), and PrintServiceAttribute (S) interfaces.



[illegible]	MONOCHROME, COLOR
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	COMPRESS, DEFLATE, GZIP, NONE
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	FIDELITY_TRUE, FIDELITY_FALSE
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

TUMBLE (= TWO_SIDED_SHORT_EDGE)

Table 12.5: Printing Attributes

As you can see, there are lots of attributes, many of which are quite specialized. The source for most of the attributes is the Internet Printing Protocol 1.1 (RFC 2911).

An earlier version of the printing API introduced the JobAttributes and PageAttributes classes, whose purpose was similar to the printing attributes covered in this section. These classes are now obsolete.

javax.print.attribute.Attribute 1.4

- Class getCategory()
  gets the category of this attribute.
- String getName()
  gets the name of this attribute.

javax.print.attribute.AttributeSet 1.4

- boolean add(Attribute attr)
  adds an attribute to this set. If the set has another attribute with the same category, that attribute is replaced by the given attribute. Returns true if the set changed as a result of this operation.
- Attribute get(Class category)
  retrieves the attribute with the given category key, or null if no such attribute exists.
- boolean remove(Attribute attr)
- boolean remove(Class category)
  remove the given attribute, or the attribute with the given category, from the set. Return true if the set changed as a result of this operation.
- Attribute[] toArray()

returns an array with all attributes in this set.

javax.print.PrintService1.4

- PrintServiceAttributeSet getAttributes()
  gets the attributes of this print service.

javax.print.DocPrintJob1.4

- PrintJobAttributeSet getAttributes()
  gets the attributes of this print job.

You have now reached the end of this long chapter covering advanced Swing and AWT features. In the final chapter, we will turn to a different aspect of Java programming: interacting, on the same machine, with “native” code in a different programming language.

Chapter 13 ■ The Foreign Functions and Memory API

While a “100% Pure Java” solution is nice in principle, there are situations in which you will want to write (or use) code in another language. Such code is usually called native code.

Particularly in the early days of Java, many people assumed that it would be a good idea to use C or C++ to speed up critical parts of a Java application. However, in practice, this was rarely useful. A presentation at the 1996 JavaOne conference showed this clearly. The developers of the cryptography library at Sun Microsystems reported that a pure Java platform implementation of their cryptographic functions was more than adequate. It was true that the code was not as fast as a C implementation would have been, but it turned out not to matter. The Java platform implementation was far faster than the network I/O. This turned out to be the real bottleneck.

Since then, the just-in-time compiler has become far better, and it is not common to turn to native code just for performance. But there are other valid reasons, such as these:

- Your application requires access to system features or devices that are not accessible through the Java platform.
- You have substantial amounts of tested and debugged code in another language.

For example, consider using an artificial intelligence library that was written in C or C++, and that uses GPUs. It may not be practical to reimplement it in Java. In such a case, a bridge between Java and native code becomes necessary.

Ever since Java 1.1, the Java platform had an API for interoperating with native C code called the Java Native Interface (JNI).

However, as you will see, there is a certain amount of tedium involved in using JNI. Java 17 introduced, as a preview feature, an API to access "foreign" functions and memory that is quite a bit more convenient than JNI. This FFM API has now become a regular feature of Java 25.

We'll start the chapter by giving you a brief flavor of JNI, and then turn to the FFM API. You should use that API for new projects. At some point in the future, JNI is likely to become deprecated.

Of course, there are drawbacks to going native. If a part of your application is written in another language, you must supply a separate native library for every platform you want to support. Code written in C or C++ offers no protection against overwriting memory through invalid pointer usage. It is easy to write native methods that corrupt your program or infect the operating system.

For that reason, since Java 22, you must opt in to native access (whether through JNI or the FFM API). If you do not use modules, start the program as

```
java --enable-native-access=ALL-UNNAMED ...
```

In a modular program, list the modules for which native access is allowed:

java --enable-native-access=*module1*, *module2* ...

13.1. Using JNI to Call C Code from a Java Program

Suppose you have a C function that does something you like and, for one reason or another, you don't want to bother reimplementing it in Java. For the sake of illustration, we'll call the C printf function with one argument.

With JNI, you need to declare a proxy method using the keyword native, and you will obviously need to place the method in a class. The result is shown in Listing 13.1.

The native keyword alerts the compiler that the method will be defined externally. Of course, native methods will contain no Java code, and the method header is followed immediately by a terminating semicolon. Therefore, native method declarations look similar to abstract method declarations.

Listing 13.1 **v2ch13/jnidemo/HelloNative.java**

```
package v2ch13.jnidemo;

class HelloNative {
   public static native int printf(String str);
}
```

In this particular example, the native method is also declared as static. Native methods can be both static and nonstatic.

You can actually compile this class, but if you try to use it in a program, the virtual machine will tell you it doesn't know how to find printf—reporting an UnsatisfiedLinkError. To implement the native code, write a corresponding C function. You must name that function exactly the way the Java virtual machine expects. Here are the rules:

1. Use the full Java method name, such as

jnidemo.HelloNative.printf.
2. Replace every period with an underscore, and append the prefix Java_. For example, Java_jnidemo_HelloNative_printf.
3. If the class name contains characters that are not ASCII letters or digits—that is, '_', '$', or Unicode characters with codes greater than \u007F—replace them with _0*xxxx*, where *xxxx* is the sequence of four hexadecimal digits of the character's Unicode value.

If you overload native methods—that is, if you provide multiple native methods with the same name—you must append a double underscore followed by the encoded argument types. For example, if you have a native method greeting() and another native method greeting(int repeat), then the first one is called Java_jnidemo_HelloNative_greeting__ and the second, Java_jnidemo_HelloNative_greeting__I.

Actually, nobody does this by hand; instead, run javac with the -h flag, providing the directory in which the header files should be placed:

```
javac -h v2ch13/jnidemo v2ch13/jnidemo/HelloNative.java
```

This command creates a header file v2ch13_jnidemo_HelloNative.h, as shown in Listing 13.2.

Listing 13.2 **v2ch13/jnidemo/v2ch13_jnidemo_HelloNative.h**

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class v2ch13_jnidemo_HelloNative */

#ifndef _Included_v2ch13_jnidemo_HelloNative
#define _Included_v2ch13_jnidemo_HelloNative
#ifdef __cplusplus
extern "C" {
#endif
```

```
/*
 * Class:     v2ch13_jnidemo_HelloNative
 * Method:    printf
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_v2ch13_jnidemo_HelloNative_printf
  (JNIEnv *, jclass, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

As you can see, this file contains the declaration of a function Java_v2ch13_jnidemo_HelloNative_printf. (The macros JNIEXPORT and JNICALL are defined in the header file jni.h. They denote compiler-dependent specifiers for exported functions that come from a dynamically loaded library.)

Now, copy the function prototype from the header file into a source file and give the implementation code for the function, as shown in Listing 13.3.

You can use C++ to implement native methods. However, you must then declare the functions that implement the native methods as extern "C". (This stops the C++ compiler from “mangling” the method name.) For example,

```
extern "C"
JNIEXPORT int JNICALL Java_v2ch13_jnidemo_HelloNative_pr
   jclass cl, . . .) { . . . }
```

Listing 13.3 **v2ch13/jnidemo/HelloNative.c**

```
#include "v2ch13_jnidemo_HelloNative.h"
#include <stdio.h>

JNIEXPORT jint JNICALL Java_v2ch13_jnidemo_HelloNative_printf(
      JNIEnv *env, jclass cl, jstring jstr) {
```

```
    const char *cstr = (*env)->GetStringUTFChars(env, jstr, NULL);
    int numchars = printf(cstr);
    (*env)->ReleaseStringUTFChars(env, jstr, cstr);
    return numchars;
}
```

We will not delve deeply into this code, but there are a couple of highlights.

Each native function receives a parameter env that is needed to access the C API for native calls. You can see an example in the call to (*env)->GetStringUTFChars(...).

The cl parameter is a descriptor of the Java class. In an instance method, you would instead get a descriptor to the object on which the native method was invoked.

Next, let's look at how to transfer strings to native methods. A native method with a String parameter receives a value of an opaque type called jstring. Strings are quite different in the two languages: In Java, they are sequences of UTF-16 code points, whereas C strings are null-terminated sequences of bytes. JNI has two sets of functions for manipulating strings: One converts Java strings to “modified UTF-8” byte sequences and another converts them to arrays of UTF-16 values. (The UTF-8, “modified UTF-8,” and UTF-16 formats were discussed in Chapter 2. Recall that the UTF-8 and “modified UTF-8” encodings leave ASCII characters unchanged, but all other Unicode characters are encoded as multibyte sequences.)

> The standard UTF-8 encoding and the “modified UTF-8” encoding differ for characters with codes higher than 0xFFFF. In the standard UTF-8 encoding, these characters are encoded as 4-byte sequences. In the “modified” encoding, each such character is first encoded as a pair of “surrogates” in the UTF-16 encoding, and then each surrogate is encoded with UTF-8, yielding a total of 6 bytes. This is clumsy, but it is a historical accident—the JVM specification was written when Unicode was still limited to 16 bits.

In our example, we call printf and therefore convert to UTF-8. Had we used the "wide" version wprintf, UTF-16 would have been appropriate.

To read the contents of an existing jstring object, use the GetStringUTFChars function. This function returns a pointer to the "modified UTF-8" characters that describe the character string. Note that a specific virtual machine is free to choose this character encoding for its internal string representation, so you might get a character pointer into the actual Java string. Since Java strings are meant to be immutable, it is very important that you treat the const seriously and do not try to write into this character array. On the other hand, if the virtual machine uses UTF-16 or UTF-32 characters for its internal string representation, this function call allocates a new memory block that will be filled with the "modified UTF-8" equivalents.

The virtual machine must know when you are finished using the string so that it can garbage-collect it. (The garbage collector runs in a separate thread, and it can interrupt the execution of native methods.) For that reason, you must call the ReleaseStringUTFChars function.

Compile the native C code into a dynamically loaded library. The details depend on your compiler.

For example, with the GNU C compiler on Linux, use these commands:

```
gcc -fPIC -I $JAVA_HOME/include -I $JAVA_HOME/include/linux -shar
   -o v2ch13/jnidemo/libHelloNative.so v2ch13/jnidemo/HelloNative.
```

With the Microsoft compiler under Windows, the command is

```
cd v2ch13/jnidemo
cl -I %JAVA_HOME%\include -I %JAVA_HOME%\include\win32 ^
   -LD HelloNative.c -FeHelloNative.dll
```

Finally, add a call to the System.loadLibrary method in your program. To ensure that the virtual machine will load the library before the first use of the class, use a static initialization

block, as in Listing 13.4.

.

Figure 13.1: Processing native code

Listing 13.4 **v2ch13/jnidemo/HelloNativeDemo.java**

```
package v2ch13.jnidemo;

class HelloNativeDemo {
   void main() {
      int result = HelloNative.printf("Hello, Native World!\n");
      IO.println("Printed %d characters.".formatted(result));
   }

   static {
      System.loadLibrary("HelloNative");
   }
}
```

Figure 13.1 gives a summary of the native code processing.

When you run this program, the message “Hello, Native World!” is displayed in a terminal window. This is the result from the C code. The next line, “Printed 21 characters.”, is produced by the Java code. We have taken the first step toward bridging the gap between the two languages!

If you run Linux, you must add the current directory to the library path. Either set the LD_LIBRARY_PATH environment variable:

```
export LD_LIBRARY_PATH=v2ch13/jnidemo:$LD_LIBRARY_PA
```

or set the java.library.path system property:

```
java -Djava.library.path=v2ch13/jnidemo v2ch13.jnidemo.Hell
```

In summary, with JNI, you follow these steps to link a native method to a Java program:

1. Declare a native method in a Java class.
2. Run javac -h to get a header file with a C declaration for the method.
3. Implement the native method in C.
4. Place the code in a shared library.
5. Load that library in your Java program.

java.lang.System1.0
■ void loadLibrary(String libname) loads the library with the given name. The library is located in the library search path. The exact method for locating the library depends on the operating system.

13.2. Using FFM to Call a Foreign Function

When using JNI, you have to write C code to access Java data structures, call the desired C functions, and to convert the result back to Java. Then you have to link the C code into platform-specific libraries. The API for accessing "foreign" functions and memory, developed under the code name "Project Panama," greatly simplifies this process.

The program in Listing 13.5 is a very simple demonstration. It simply calls the printf function. No additional C code is necessary. The function is invoked through a MethodHandle object.

To access C functions, you first create a Linker object. The Linker can produce a Java method handle for a C function, given its address and type description.

The API has methods for translating Java objects into memory segments that you can pass to the C function. In the demo, you need to provide a char* with the characters from a Java String.

The memory is allocated in an Arena, so that its deallocation can be managed. Here, we use a confined arena. It is autocloseable, and the close method deallocates all allocated memory.

To compile and run the demo program, use these commands:

```
javac v2ch13/ffmdemo/HelloForeign.java
java --enable-preview --enable-native-access=ALL-UNNAMED v2ch13
```

As you can see, this API is more Java-centric than JNI. All the conversion from Java to C is done in Java code, and you simply invoke the C library function from Java. In the following sections, you will see all the details about linkers, method handles, arenas, and memory segments.

Listing 13.5 **v2ch13/ffmdemo/HelloForeign.java**

```
package v2ch13.ffmdemo;

import java.lang.invoke.*;
import java.lang.foreign.*;

public class HelloForeign {
   void main() throws Throwable {
      Linker linker = Linker.nativeLinker();
      MethodHandle printf = linker.downcallHandle(
         linker.defaultLookup().findOrThrow("printf"),
         FunctionDescriptor.of(ValueLayout.JAVA_INT, ValueLayout.ADDRESS));

      try (Arena arena = Arena.ofConfined()) {
         MemorySegment str = arena.allocateFrom("Hello, Foreign World!\n");
         int result = (int) printf.invoke(str);
         System.out.printf("Printed %d characters.%n", result);
      }
   }
}
```

13.3. Arenas

When interfacing with foreign functions, the Java part of the program needs to work with off-heap memory; that is, memory that was not allocated with a call to new.

An arena manages the lifetime of off-heap memory segments. You have seen one example in the preceding demo program.

```
try (Arena arena = Arena.ofConfined()) {
   MemorySegment str = arena.allocateFrom(. . .);
   . . .
} // Arena closed
```

The Arena class implements AutoCloseable, and the arena instance is closed when the try block ends. Any allocated segments become invalid afterwards. Attempting to access them causes an exception.

Here, a confined arena is used. The memory segments that a confined arena allocates can only be used in the arena's owner thread, which is usually the thread in which the arena was created. The arena must also be closed in that thread. Otherwise an exception is thrown.

If memory segments must be shared by multiple threads, you cannot use a confined arena. One option is a shared arena, which is alive until you call close on it. It is up to you to close the arena at the appropriate time.

Alternatively, you can use an automatic arena. The garbage collector will eventually deallocate memory segments when they, and the arena, are no longer referenced. This is convenient, but less predictable. You cannot close an automatic arena.

```
MemorySegment segment = Arena.ofAuto().allocate(. . .);
. . .
segment = null; // Memory eventually deallocated
```

In this example, the automatic arena is only used once, to allocate a single segment.

Finally, the global arena is never closed, and the memory segments that it allocates are never deallocated.

You cannot deallocate individual memory segments. To deallocate a memory segment, you need to close the (confined or shared) arena that allocated it.

Table 13.1 summarizes the characteristics of the four arena types.

Type	Accessible from multiple threads	How closed
Confined		By calling close
Shared	✓	By calling close
Automatic	✓	By garbage collector
Global	✓	Never

Table 13.1: Arena Types

java.lang.foreign.Arena[22]

- static void ofConfined()
- static void ofShared()
- static void ofAuto()
  Yields a confined, shared, or automatic arena.

13.4. Memory Segments

A memory segment is a contiguous block of memory. It can be on the Java heap, managed by the garbage collector, or native, allocated off the Java heap. An off-heap memory segment has a fixed physical address. A heap segment, just like any Java object, has a reference through which it can be located, but no fixed address since the garbage collector may have moved it. For use with foreign functions, we are generally interested in off-heap memory segments.

You use an arena object to allocate an off-heap segment. As you already have seen, the call

```
String javaString = . . .;
MemorySegment cString = arena.allocateFrom(javaString);
```

encodes a Java string into UTF-8 and allocates a memory segment that holds the bytes, including a null terminator.

To allocate an array, you need to specify the element layout and the number of elements. You will see how to use the MemoryLayout interface in Section 13.5. For now, we will make use of the layout constants defined in the ValueLayout class. For example, here is how to allocate an array of float values:

```
MemorySegment audioIn = arena.allocate(ValueLayout.JAVA_FLOAT
```

If you already have the values in an array, use the allocateFrom method:

```
float[] samples = . . .;
MemorySegment audioOut = arena.allocateFrom(ValueLayout.JAVA_
```

When a memory segment is passed to a C function, the function receives the address of the segment. It can then process the contents of the string or array.

Conversely, suppose a C function has filled a memory segment with data. To turn a null-terminated UTF-8 string to a Java string, you pass an offset of 0 to the getString method:

```
String javaString = cString.getString(0);
```

To read or write at an array index, provide the memory layout and index:

```
float sample = audioIn.getAtIndex(ValueLayout.JAVA_FLOAT, i);
audioIn.setAtIndex(ValueLayout.JAVA_FLOAT, i, 0f);
```

You can also get a Java array with all values:

```
float[] samples = audioIn.toArray(ValueLayout.JAVA_FLOAT);
```

The elements method yields a stream of memory segments, one for each element. You can map each of them into a value and then process the stream further.

```
DoubleStream samples = audioIn.elements(ValueLayout.JAVA_FLOAT
   .mapToDouble(m -> m.get(ValueLayout.JAVA_FLOAT, 0));
```

The get method is like getAtIndex, but it takes a byte offset.

Note that the segment does not retain any information about the layout of the data inside. You need to provide the layout whenever you access or update the data.

java.lang.foreign.SegmentAllocator22

- MemorySegment allocate(MemoryLayout layout)
  allocates off-heap memory with the given layout.
- MemorySegment allocate(MemoryLayout elementLayout, long count)
  allocates an off-heap array with count elements of the given layout.
- MemorySegment allocateFrom(ValueLayout.of*Xxx* elementLayout, *xxx* elements)
  Allocates off-heap memory holding the given primitive type elements.
- MemorySegment allocateFrom(String str)
  allocates off-heap memory holding the null-terminated string with the UTF-8 encoding of str.
- MemorySegment allocate(long byteSize, long byteAlignment)
  allocates an off-heap memory segment of the given size and alignment constraint.
- Stream<MemorySegment> elements(MemoryLayout elementLayout)
  yields a sequential stream of slices with the given layout, taken from this memory segment.

java.lang.foreign.MemorySegment22

- String getString(long offset)
  converts the null-terminated UTF-8 string at the given offset to a Java string.
- *xxx* getAtIndex(ValueLayout.of*Xxx* layout, long index)
- void setAtIndex(ValueLayout.of*Xxx* layout, long index, *xxx* value)
  read or write a primitive type value at the given array index.
- *xxx* get(ValueLayout.of*Xxx* layout, long offset)
- void set(ValueLayout.of*Xxx* layout, long offset, *xxx* value)
  read or write a primitive type value at the given offset.
- *xxx*[] toArray(ValueLayout.of*Xxx* elementLayout)
  copies the contents of this memory segment into a primitive type array.

13.5. Memory Layout

In the preceding section, you have seen how to work with memory segments that hold UTF-8 strings and arrays of primitive type values.

In C, you use struct for more complex data. For example, the libsndfile library for processing sound files (https://github.com/libsndfile) yields a value of the following structure when opening a sound file:

```
struct SF_INFO {
   long frames;
   int samplerate;
   int channels;
   int format;
   int sections;
   int seekable;
} ;
```

You can define a layout for this structure as follows:

```
import static java.lang.foreign.ValueLayout.*;
. . .
```

```
StructLayout SF_INFO = MemoryLayout.structLayout(
    JAVA_LONG.withName("frames"),
    JAVA_INT.withName("samplerate"),
    JAVA_INT.withName("channels"),
    JAVA_INT.withName("format"),
    JAVA_INT.withName("sections"),
    JAVA_INT.withName("seekable")
);
```

When matching up the C and Java types, the byte lengths need to match. But unsigned C types are mapped to their sized Java equivalents. If you process those values in Java, you need t to handle them as unsigned values yourself.

> The expression Linker.nativeLinker().canonicalLayouts() yields a map from common type names to memory layouts. For example, on a 64-bit platform, "size_t" is mapped to a MemoryLayout with byte size 8, so you know to use a JAVA_LONG to represent the C type size_t in Java.
>
> The following types are mapped: bool, char, short, int, long, long long, float, double, size_t, wchar_t, void*.

Suppose you have a memory segment info with the SF_INFO layout that was just declared. Here is how you can retrieve one of the components:

```
int channels = info.get(JAVA_INT,
    SF_INFO.byteOffset(PathElement.groupElement("channels")));
```

You have already seen the get method, which receives a memory layout and a byte offset. The StructLayout can determine the offset of each element. In this case, the offset is 12, since the "channels" element is preceded by elements of size 8 and 4.

In order to deal with nested structures, there are a number of interfaces to describe layouts—see Figure 13.2. The ValueLayout interface has subtypes ValueLayout.OfInt,

ValueLayout.ofLong, and so on, one for each primitive type. The constants JAVA_INT, JAVA_LONG, and so on in the ValueLayout interface are instances of these types. The AddressLayout holds a memory address.

.

Figure 13.2: The MemoryLayout Hierarchy

As you have seen, a StructLayout describes the layout of C structures. Similarly, the UnionLayout describes C unions. This is not common, but if you need to deal with unions, you shouldn't have any trouble using the UnionLayout interface. It is completely analogous to StructLayout.

The PaddingLayout describes padding bytes. You sometimes need to insert them to ensure that subsequent elements are properly aligned.

Let us use the SequenceLayout to model an array of structures:

```
long count = . . .;
SequenceLayout arrayOfPointLayout = MemoryLayout.sequenceLay
   MemoryLayout.structLayout(
      JAVA_FLOAT.withName("x"),
      JAVA_FLOAT.withName("y")));
```

To access an individual x- or y-component of the point with index i, we need to compute the byte offset:

```
long offset = arrayOfPointLayout.byteOffset(
   PathElement.sequenceElement(i), PathElement.groupElement("y")
```

The byteOffset method receives a sequence of MemoryLayout.PathElement objects that navigate to the selected value.

Now, given a memory segment with that layout, we can get or set the value:

```
MemorySegment arrayOfPoints = Arena.global().allocate(arrayOfPoi
```

```
arrayOfPoints.set(JAVA_FLOAT, offset, yvalue);
```

There is more to know about memory layouts and accessing values in memory segments, but what you have seen now suffices to pass complex data between Java and foreign functions. We will take that up in the following section. For more information, see Section 13.8.1.

java.lang.foreign.MemoryLayout22

- static StructLayout structLayout(MemoryLayout... elements)
- static UnionLayout unionLayout(MemoryLayout... elements)
  create the layout for a struct or union with the given elements.
- static SequenceLayout sequenceLayout(long elementCount, MemoryLayout elementLayout)
  creates the layout for an array of the given length, whose elements have the given layout.
- static PaddingLayout unionLayout(long byteSize)
  creates the layout for padding with the given number of bytes.
- long byteOffset(MemoryLayout.PathElement... elements)
  yields the offset in this layout for the first value specified by the given path.

java.lang.foreign.MemoryLayout.PathElement22

- static MemoryLayout.PathElement groupElement(String name)
  yields a path to the element with the given name in a group layout.
- static MemoryLayout.PathElement sequenceElement(long index)
  yields a path to the element with the given index in a sequence layout.

java.lang.foreign.ValueLayout[22]

- ValueLayout withName(String name)
  yields a copy of this layout with the given name.

13.6. Looking Up and Invoking Foreign Functions

When compiling C code, a linker carries out the essential step of combining the compiled files with precompiled code of library functions, producing an executable file.

The Linker abstraction in the FFM API doesn't quite do the same thing. It looks up library functions and prepares MethodHandle objects to invoke them.

Actually, the "looking up" part is delegated to a SymbolLookup instance. To look up functions in a handful of standard libraries, use a default lookup:

```
Linker linker = Linker.nativeLinker(); // The linker for the underlying
SymbolLookup stdlib = linker.defaultLookup();
```

The libraries that the default lookup knows about depend on the platform. On Linux, they include libc and libm. This lookup suffices for well-known functions such as printf and malloc.

For use with a specialized library, create a lookup as follows:

```
SymbolLookup sndfile = SymbolLookup.libraryLookup(
   "/lib/x86_64-linux-gnu/libsndfile.so", arena);
```

You specify the path to the library file (as a string or Path object), and an arena. The libraryLookup method loads the library. When the arena is closed, the library is unloaded.

There is also a loader lookup, returned by SymbolLookup.loaderLookup(), that looks up functions

---

in libraries loaded through calls to System.load and System.loadLibrary.

Now you are ready to look up functions:

```
MemorySegment mallocAddr = stdlib.findOrThrow("malloc");
MemorySegment sfOpenAddr = sndfile.findOrThrow("sf_open");
```

The findOrThrow method throws a NoSuchElementException if the function was not found. There is also a find method that returns an Optional<MemorySegment> instead. The memory segment contains the address of the function to be invoked.

Many programmers look at the Java API as a model for good design, but in this particular instance, you might want to look twice.

The find method returns an Optional because surely that is the right thing to do when a lookup might fail. Or is it?

If you are in a similar situation when designing an API, consider whether you want to burden its users with Optional results. Of course, a lookup might fail. Java has a perfectly good mechanism for that—an exception.

An Optional is appropriate if there is a plausible action to take upon failure, such as substituting a default value, or skipping further processing. But if a library function isn't found, there is unlikely to be a local remedy, and an exception is the best outcome.

The findOrThrow method was added later to avoid the tedium of calling

```
MemorySegment mallocAddr = stdlib.find("malloc").orElseThr
```

It dispenses with the Optional, which is good. But what about the orThrow suffix? Fortunately, few exception-throwing methods have such a suffix in their name. The notable exception is Optional.orElseThrow, which was

given a scary name so that programmers would consider alternatives, such as orElse or map.

When you find yourself implementing a method whose failure is unlikely to be curable locally, don't return an Optional (and of course, don't return null). Just throw an exception. Without uglifying the method name.

Having found the memory address of the function to be invoked, we are ready to create a method handle for the invocation. A method handle is an object through which you can provide arguments, invoke a method, and obtain the returned value. It is similar to a Method object in the java.lang.reflect package, but it can also invoke native methods.

To get the method handle, provide the parameter and return types:

```
// void* malloc(size_t)
MethodHandle mallocHandle = linker.downcallHandle(mallocAddr,
   FunctionDescriptor.of(ADDRESS, JAVA_LONG));

// SNDFILE* sf_open(const char *path, int mode, SF_INFO *sfinfo)
MethodHandle sfOpenHandle = linker.downcallHandle(sfOpenAddr,
   FunctionDescriptor.of(ADDRESS, ADDRESS, JAVA_INT, ADDRESS));
```

For a function with return type void, you use FunctionDescriptor.ofVoid instead:

```
// void free(void *ptr)
MethodHandle freeHandle = linker.downcallHandle(
   stdlib.findOrThrow("free"),
   FunctionDescriptor.ofVoid(ADDRESS));
```

Here, we ask for a downcall handle for calling a C function from Java code. In Section 13.7, you will see how to use memory handles for calling a Java method from C.

When passing a pointer to a C function, or receiving a pointer result, you use ValueLayout.ADDRESS. The type

---

of the memory block is immaterial.

You must describe the function return type and parameter types accurately. The linker cannot verify this information. If you provide incorrect information, the virtual machine may crash.

Finally, we are ready to invoke the method.

```
MemorySegment block = mallocHandle.invoke(blockSize);

MemorySegment info = arena.allocate(SF_INFO);
MemorySegment cInputPath = arena.allocateFrom(inputFile);
MemorySegment inFile = (MemorySegment) sf_open.invoke(cInputP
```

When a foreign function returns a pointer, it is wrapped into a zero-length memory segment. To read from the segment, you need some external knowledge of its size and structure.

For example, you know the size of the memory block allocated by malloc, so you can set

```
block = block.reinterpret(blockSize);
```

Clearly, this is a potentially unsafe operation. If you use any memory segment outside its actual bounds, you can corrupt unrelated memory or crash the virtual machine.

The jextract tool, which you can find at https://github.com/openjdk/jextract, automatically generates memory layouts and wrappers for foreign calls from C header files.

You now know enough about the FFM API to write nontrivial programs that call a native library. The program in Listing 13.6 is a typical example. It uses the libsndfile library to read a sound file, reverses the samples, and writes the result to another file.

The program first sets up the layout for the SF_INFO structure and memory handles for four C functions:

```
SNDFILE* sf_open(const char *path, int mode, SF_INFO *sfinfo)
int sf_close(SNDFILE *sndfile)
sf_count_t sf_read_float(SNDFILE *sndfile, float *ptr, sf_count_t items
sf_count_t sf_write_float(SNDFILE *sndfile, const float *ptr, sf_count_
```

Then the program opens the input file and retrieves the number of frames and channels from the SF_INFO structure.

Next, the call to sf_read_float fills a pre-allocated memory segment with the sample data.

For ease of processing in Java, the memory segment is converted to a Java array. After processing, the data is written to the output file.

Listing 13.6 **v2ch13/sound/SoundFileReverser.java**

```
package v2ch13.sound;

import module java.base;

import static java.lang.foreign.ValueLayout.*;

class SoundFileReverser {
   static final int SFM_READ = 0x10;
   static final int SFM_WRITE = 0x20;

   void main(String[] args) throws Throwable {
      String inputFile = args.length > 0 ? args[0] : "sound/input.mp3";
      String outputFile = args.length > 1 ? args[1] : "sound/output.mp3";

      Linker linker = Linker.nativeLinker();
      SymbolLookup sndfile = SymbolLookup.libraryLookup(
         "/lib/x86_64-linux-gnu/libsndfile.so", Arena.ofAuto());

      StructLayout SF_INFO = MemoryLayout.structLayout(
         JAVA_LONG.withName("frames"),
         JAVA_INT.withName("samplerate"),
```

```
    JAVA_INT.withName("channels"),
    JAVA_INT.withName("format"),
    JAVA_INT.withName("sections"),
    JAVA_INT.withName("seekable")
);

MethodHandle sf_open = linker.downcallHandle(
    sndfile.findOrThrow("sf_open"),
    FunctionDescriptor.of(ADDRESS, ADDRESS, JAVA_INT, ADDRESS)
);

MethodHandle sf_close = linker.downcallHandle(
    sndfile.findOrThrow("sf_close"),
    FunctionDescriptor.of(JAVA_INT, ADDRESS)
);

MethodHandle sf_readf_float = linker.downcallHandle(
    sndfile.findOrThrow("sf_readf_float"),
    FunctionDescriptor.of(JAVA_LONG, ADDRESS, ADDRESS, JAVA_LONG)
);

MethodHandle sf_writef_float = linker.downcallHandle(
    sndfile.findOrThrow("sf_writef_float"),
    FunctionDescriptor.of(JAVA_LONG, ADDRESS, ADDRESS, JAVA_LONG)
);

try (Arena arena = Arena.ofConfined()) {
    MemorySegment info = arena.allocate(SF_INFO);
    MemorySegment cInputPath = arena.allocateFrom(inputFile);

    // Open sound file
    MemorySegment inFile = (MemorySegment) sf_open.invoke(cInputPath
    if (inFile == MemorySegment.NULL)
        throw new RuntimeException("sf_open: Failed for " + inputFile);
    long frames = info.get(JAVA_LONG,
        SF_INFO.byteOffset(PathElement.groupElement("frames")));
    int channels = info.get(JAVA_INT,
        SF_INFO.byteOffset(PathElement.groupElement("channels")));
    long totalSamples = frames * channels;

    // Allocate native memory for audio samples
    MemorySegment audioIn = arena.allocate(JAVA_FLOAT, totalSamples);

    // Read samples
```

```
        long actualFrames = (long) sf_readf_float.invoke(
            inFile, audioIn, frames);
        if (frames != actualFrames)
            throw new RuntimeException(
                "sf_readf_float: Expected %d frames but read %d frames."
                .formatted(frames, actualFrames));

        // Copy native array to Java and process
        float[] audioSamples = audioIn.toArray(JAVA_FLOAT);

        // Reverse the frames
        for (int f = 0; f < frames / 2; f++) {
            for (int c = 0; c < channels; c++) {
                int i = f * channels + c;
                int j = (int)((frames - 1 - f) * channels + c);
                float temp = audioSamples[i];
                audioSamples[i] = audioSamples[j];
                audioSamples[j] = temp;
            }
        }

        // Write to sound file
        MemorySegment cOutputPath = arena.allocateFrom(outputFile);
        MemorySegment outFile = (MemorySegment) sf_open.invoke(
            cOutputPath, SFM_WRITE, info);
        if (outFile == MemorySegment.NULL)
            throw new RuntimeException("sf_open: Failed for " + outputFile);

        // Copy Java array to native and write
        MemorySegment audioOut = arena.allocateFrom(JAVA_FLOAT, audioSa

        actualFrames = (long) sf_writef_float.invoke(outFile, audioOut, frames);
        if (frames != actualFrames)
            throw new RuntimeException(
                "sf_writef_float: Expected to write %d frames but wrote %d frames.
                .formatted(frames, actualFrames));

        sf_close.invoke(inFile);
        int error = (int) sf_close.invoke(outFile);
        if (error != 0)
            throw new RuntimeException("sf_close: Failed with error code " + err
        IO.println("Reversed " + inputFile + " to  " + outputFile);
    }
}
```

}

java.lang.foreign.Linker22

- static Linker nativeLinker()
  yields a linker for the underlying platform.
- SymbolLookup defaultLookup()
  yields a symbol lookup for a default set of libraries.
- Map<String, MemoryLayout> canonicalLayouts()
  yields a map that holds layouts for data types used by the underlying platform.
- MethodHandle downcallHandle(MemorySegment address, FunctionDescriptor function, Linker.Option... options)
  yields a method handle for invoking the function with the given address and function descriptor.

java.lang.foreign.SymbolLookup22

- static SymbolLookup libraryLookup(String name, Arena arena)
- static SymbolLookup libraryLookup(Path path, Arena arena)
  load the library with the given name or path and yields a symbol lookup for the library symbols. The library is unloaded when the given arena is closed.
- Optional<MemorySegment> find(String name)
- MemorySegment findOrThrow(String name)
  yield the symbol with the given name, returning an empty Optional or throwing a NoSuchElementException if the symbol is not found.

java.lang.foreign.FunctionDescriptor22

- static FunctionDescriptor ofVoid(MemoryLayout... argLayouts)
- static FunctionDescriptor of(MemoryLayout resLayout, MemoryLayout... argLayouts)

yield a function descriptor with the given argument layouts and return type void, or the given return layout.

java.lang.invoke.MethodHandle7

- Object invoke(Object... args)
  invokes the method handle with the given arguments.

java.lang.foreign.MemorySegment22

- MemorySegment reinterpret(long newSize)
  yields a copy of this memory segment with the given size.

13.7. Callbacks

Sometimes, you call a foreign function that wants to call another function when an event occurs; for example, when input is detected at a microphone or camera. I would prefer to demonstrate this with a realistic example, but there is a fair amount of boilerplate and configuration required. Instead, here is an example with the qsort function from the standard library:

```
void qsort(void *base, size_t nmemb, size_t size,
   int (*compar)(const void *, const void *));
```

The last parameter is a pointer to a function that is being called whenever two elements are compared.

We want the following Java method to be called back:

```
static int comp(MemorySegment a, MemorySegment b) {
   return Float.compare(a.get(JAVA_FLOAT, 0), b.get(JAVA_FLOAT, 0));
}
```

> As already mentioned, keep in mind that this is a toy example to show the mechanics of a callback. One would not normally split up this work between Java and C.

The following code snippet gets a method handle to the foreign function in the usual way:

```
Linker linker = Linker.nativeLinker();
MethodHandle qsort = linker.downcallHandle(
  linker.defaultLookup().findOrThrow("qsort"),
  FunctionDescriptor.of(ADDRESS, JAVA_LONG, JAVA_LONG, ADDRE
```

When invoking the function, we supply an array, its length, the size of each element in bytes, and the address of the callback.

To produce that address, first get a MemoryHandle to the static Java method:

```
MethodHandle compHandle = MethodHandles.lookup()
  .findStatic(UpcallDemo.class, "comp",
    MethodType.methodType(
      int.class, MemorySegment.class, MemorySegment.class));
```

This has nothing to do with foreign functions. One can get a method handle to any Java method, and invoke it later.

The linker can generate stub code for a C function that then invokes the method handle of the callback:

```
MemorySegment compStub = linker.upcallStub(
  compHandle,
  FunctionDescriptor.of(JAVA_INT,
    ADDRESS.withTargetLayout(ValueLayout.JAVA_FLOAT),
    ADDRESS.withTargetLayout(ValueLayout.JAVA_FLOAT)),
  arena);
```

Note the withTargetLayout calls. They configure the memory layout so that the linker can generate code for configuring the MemorySegment instances that the Java callback receives. If you don't do that, the calls to a.get(JAVA_FLOAT, 0) and b.get(JAVA_FLOAT, 0) in the Java callback will fail because the

memory segments a and b will have size zero.

Finally, you are ready to call the foreign function with the callback address:

```
float[] samples = . . .;
MemorySegment csamples = arena.allocateFrom(JAVA_FLOAT, samp
qsort.invoke(csamples, samples.length, JAVA_FLOAT.byteSize(), comp
```

You will find the complete program in Listing 13.7

Listing 13.7 **v2ch13/upcall/UpcallDemo.java**

```
package v2ch13.upcall;

import module java.base;

import static java.lang.foreign.ValueLayout.*;

class UpcallDemo {
   static int comp(MemorySegment a, MemorySegment b) {
      return Float.compare(a.get(JAVA_FLOAT, 0), b.get(JAVA_FLOAT, 0));
   }

   void main() throws Throwable {
      Linker linker = Linker.nativeLinker();
      MethodHandle qsort = linker.downcallHandle(
         linker.defaultLookup().findOrThrow("qsort"),
         FunctionDescriptor.ofVoid(ADDRESS, JAVA_LONG, JAVA_LONG, ADDRES

      MethodHandle compHandle = MethodHandles.lookup()
         .findStatic(UpcallDemo.class, "comp",
            MethodType.methodType(
               int.class, MemorySegment.class, MemorySegment.class));

      MemorySegment compStub = linker.upcallStub(
         compHandle,
         FunctionDescriptor.of(JAVA_INT,
            ADDRESS.withTargetLayout(ValueLayout.JAVA_FLOAT),
            ADDRESS.withTargetLayout(ValueLayout.JAVA_FLOAT)),
         Arena.ofAuto());
```

```
      float[] samples = new float[100];
      RandomGenerator generator = RandomGenerator.getDefault();
      for (int i = 0; i < samples.length; i++)
         samples[i] = generator.nextFloat();
      try (Arena arena = Arena.ofConfined()) {
         MemorySegment csamples = arena.allocateFrom(JAVA_FLOAT, samples)
         qsort.invoke(csamples,
            samples.length, JAVA_FLOAT.byteSize(), compStub);
         samples = csamples.toArray(JAVA_FLOAT);
         IO.println(Arrays.toString(samples));
      }
   }
}
```

java.lang.foreign.Linker22

- MemorySegment upcallStub(MethodHandle target, FunctionDescriptor function, Arena arena, Linker.Option... options)
  generates function code, and yields a function pointer through which native code can call the given method handle, using the argument and return layouts given by the function descriptor. The memory for the function code is reclaimed when the arena closes.

java.lang.invoke.MethodHandles7

- static MethodHandle.Lookup lookup()
  yields a lookup object for looking up method handles.

java.lang.invoke.MethodHandle.Lookup7

- MethodHandle findStatic(Class<?> refc, String name, MethodType type)
  finds a method handle for a static method of the given class with the given name and type.

java.lang.invoke.MethodType7

- static MethodType methodType(Class<?> rtype, Class<?> ptype0, Class<?>... ptypes)
  yields a method type with the given return type and parameter types.

13.8. Advanced Topics

You have now seen enough about the FFM API to write nontrivial programs. The API provides solutions for a number of more complex situations that we will examine in the following sections.

13.8.1. Accessing Memory With Variable Handles

A variable handle allows getting or setting a single value, or multiple values that are specified by coordinates. These could be array indices or map keys, or any other objects that describe the location of a value.

```
VarHandle vh = . . .;
Object value = vh.get(coord1, coord2, . . .);
vh.set(coord1, coord2, . . ., newValue);
```

In Section 13.5, you have seen how to access elements in a struct by computing the element offset from a layout. We looked at the example of the layout of the SF_INFO structure that has a channels element.

```
long channelsOffset = SF_INFO.byteOffset(PathElement.groupElemer
int channels = info.get(JAVA_INT, channelsOffset);
```

Instead of computing offsets, you can use variable handles. For example, here is a variable handle to the channels element:

```
VarHandle channelsHandle = SF_INFO.varHandle(PathElement.grou
```

This variable handle has a single “coordinate”—the memory

segment holding the structure. Here is how you look inside a memory segment info to read the field value:

```
int channels = (int) channelsHandle.get(info);
```

Using a variable handle isn't any simpler than computing a byte offset, but it is a bit safer. After all, it is easy enough to accidentally use the wrong offset.

Now consider an array layout:

```
long count = . . .;
StructLayout pointLayout = MemoryLayout.structLayout(
   JAVA_FLOAT.withName("x"),
   JAVA_FLOAT.withName("y"));
SequenceLayout arrayOfPointLayout = MemoryLayout.sequenceLay
```

Here is a variable handle for the x-coordinate of a point in the array:

```
VarHandle xHandle = arrayOfPointLayout.varHandle(
   PathElement.sequenceElement(),
   PathElement.groupElement("x"));
```

The PathElement.sequenceElement() call returns an open path element which gives rise to another coordinate in the variable handle. Now you can access the ith x-coordinate as:

```
MemorySegment pointSegment = . . .;
long i = . . .;
float xi = (float) xHandle.get(pointSegment, i);
```

Now suppose you have an array of pointers, each to a point structure.

```
MemoryLayout arrayOfPointPointersLayout = MemoryLayout.seque
   ValueLayout.ADDRESS.withTargetLayout(pointLayout));
```

Note the withTargetLayout method that attaches the layout of the memory segment to which the address points. We did not do that in the past with ADDRESS because we could not make use of the type information. But there is a special path element

for following a pointer:

```
VarHandle xHandle = arrayOfPointPointersLayout.varHandle(
   PathElement.sequenceElement(),
   PathElement.dereferenceElement(),
   PathElement.groupElement("x"));
```

A MemoryLayout can also produce a method handle for computing byte offsets, which is potentially convenient for paths with open path elements.

```
MethodHandle xByteOffsetHandle = arrayOfPointLayout.byteOffsetH
   PathElement.sequenceElement(),
   PathElement.groupElement("x"));
```

Now you can compute an offset like this:

```
long offset = (long) xByteOffsetHandle(0, i);
```

The first argument is the base offset, usually zero.

Of course, this doesn't work when the path doesn't have indirections.

13.8.2. Calling a Foreign Varargs Function

In C, functions with variable arguments use a special calling convention. The classic example is

```
int printf(const char *format, ...);
```

In order to call such a function, the FunctionDescriptor must know where the fixed arguments end and the variable arguments start.

You must make a separate method handle for each pattern of arguments. For example, suppose you’d like to call

```
printf("Hello, %s! Next year, you'll be %d\n", name, age);
```

You will be passing the format string, another string, and an integer.

```
MethodHandle printf = linker.downcallHandle(
   linker.defaultLookup().findOrThrow("printf"),
      FunctionDescriptor.of(JAVA_INT, ADDRESS, ADDRESS, JAVA_INT)
      Linker.Option.firstVariadicArg(1));
```

The linker option specfies the index of the first variable parameter.

13.8.3. Capturing errno

Some C functions signal an error condition by placing an integer error code in a location that can be retrieved with the errno macro.

To capture this value, use another linker option:

```
MethodHandle fopen = linker.downcallHandle(
   linker.defaultLookup().findOrThrow("fopen"),
   FunctionDescriptor.of(ADDRESS, ADDRESS, ADDRESS),
   Linker.Option.captureCallState("errno"));
```

Before calling the foreign function, allocate a memory segment for the capture state:

```
StructLayout captureStateLayout = Linker.Option.captureStateLayou
MemorySegment captureState = arena.allocate(captureStateLayout);
```

Pass that memory segment as the first argument when invoking the method handle:

```
var result = (MemorySegment) fopen.invoke(captureState,
   arena.allocateFrom("woozle.txt"),
   arena.allocateFrom("rw"));
```

Upon return, you can read errno from the capture state:

```
long errnoOffset = capturedStateLayout.byteOffset(PathElement.grou
int errno = (int) capturedState.get(JAVA_INT, errnoOffset);
```

Or, with a variable handle:

```
VarHandle errnoHandle = captureStateLayout.varHandle(PathEleme
```

```
int errno = (int) errnoHandle.get(capturedState, 0);
```

To get the values that can be captured, look at

```
List<MemoryLayout> layouts = Linker.Option.captureStateLay
```

On Linux, you currently only get errno.

java.lang.foreign.MemoryLayout22

- VarHandle varHandle(MemoryLayout.PathElement... elements)
  yields a variable handle for looking up the value to which the path leads.
- MethodHandle byteOffsetHandle(MemoryLayout.PathElement... elements)
  yields a method handle for computing the byte offset of the value to which the path leads.

java.lang.foreign.MemoryLayout.PathElement22

- static MemoryLayout.PathElement sequenceElement()
  yields an “open” path element for selecting one of the elements in the sequence.
- static MemoryLayout.PathElement dereferenceElement()
  yields a path element that selects the target layout of an address layout.

java.lang.foreign.AddressLayout22

- AddressLayout withTargetLayout(MemoryLayout layout)
  yields a copy of this layout with the given target layout; that is, the layout of the memory segment to which this address points.

java.lang.foreign.Linker.Option22

- static Linker.Option firstVariadicArg(int index)
  yields a linker option describing the index of the first variadic argument of the function to be linked in a downcall.
- static Linker.Option captureCallState(String... capturedState)
  yields a linker option directing a downcall invocation to capture the state items with the given names.
- static StructLayout captureStateLayout()
  yields the layout for the capture state.

You have now reached the end of the second volume of Core Java, completing a long journey in which you encountered many advanced APIs. I started out with topics that every Java programmer needs to know: streams, XML, networking, databases, and internationalization. I concluded with very technical chapters on security, annotation processing, advanced graphics, and native methods. I hope that you enjoyed your tour through the vast breadth of the Java APIs, and that you will be able to apply your newly gained knowledge in your projects.

Index

Symbols

≤ operator 7.5.2
method
of LocalTime 6.7
!= operator (SQL) 5.2
(number sign)
in decimal formatting 7.2.2
in message formatting 7.5.2
in URLs 4.3.1
#PCDATA (DTDs) 3.4.1
$ (dollar sign)
in native method names 13.1
in regular expressions 2.7.1, 2.7.6
% (percent sign)
in locales 7.2.1
in SQL 5.2, 5.5.3
in URLs 4.3.3
¤ 7.2.2
“Seek forward only” mode (ImageInputStream) 12.6.3
€ 7.2.3, 7.7.3
& (ampersand)
in CDATA sections 3.2
in entity references 3.2
parsing 3.4.1
&#, &#x, in character references 3.2
&& operator 2.7.1
&, entity reference 3.2
', entity reference 3.2
>, entity reference 3.2
<, entity reference 3.2
", entity reference 3.2
> (right angle bracket) 3.2
< (left angle bracket)
in CDATA sections 3.2
in message formatting 7.5.2
parsing 3.4.1
<!--. . .-->, <?. . .?>, <![CDATA[. . .]]>, in XML 3.2
<> operator (SQL) 5.2

<. . .> (angle brackets, in regular expressions) 2.7.1
". . .", in XML 3.1
' 3.2
'. . .', in SQL 5.2
(. . .) (parentheses)
in regular expressions 2.7.1, 2.7.4
* (asterisk)
echo character 11.3.3
in DTDs 3.4.1
in glob patterns 2.3.7
in regular expressions 2.7.1, 2.7.4
+ (plus sign)
in DTDs 3.4.1
in regular expressions 2.7.1
in URLs 4.3.3
operator, identity for 1.10
, (comma)
decimal 7.1.1, 7.2.1, 7.2.2
in DTDs 3.4.1
- (minus sign)
in regular expressions 2.7.1
in URLs 4.3.3
. (period)
decimal 7.1.1, 7.2.1, 7.2.2
in regular expressions 2.7.1, 2.7.7
in URLs 4.3.3
.., in paths 2.3.1
.class extension 9.1
/ (slash)
in paths 2.1.3, 2.3.1
in URLs 4.3.1
1931 CIE XYZ color specification 12.6.4
2D graphics. See Java 2D API
2D shapes 10.3.1
: (colon)
in classpath 5.3.2
in text files 2.1.7
in URLs 4.3.1
; (semicolon)
in classpath 5.3.2
in decimal formatting 7.2.2
in SQL 5.3.3
= operator (SQL) 5.2
? (question mark)
in DTDs 3.4.1
in glob patterns 2.3.7
in prepared queries 5.5.1
in regular expressions 2.7.1
in URLs 4.3.3
@ (at sign)

in URIs 4.3.1
in XPath 3.5
@Override annotation 2.6.5
@Resource annotation 5.10
@Serial annotation 2.6.4, 2.6.5
@SuppressWarnings annotation 2.6.7
[(array), type code 2.6.2
[. . .] (brackets)
in DOCTYPE declaration 3.4.1
in glob patterns 2.3.7
in regular expressions 2.7.1
in XPath 3.5
\ (backslash)
in glob patterns 2.3.7
in paths 2.1.3, 2.3.1
in regular expressions 2.7.1, 2.7.6
\0, in regular expressions 2.7.1
\\|, in regular expressions 2.1.7
\a, \A, in regular expressions 2.7.1
\A, in regular expressions 2.7.1
\b, \B, in regular expressions 2.7.1
\d, \D, in regular expressions 2.7.1
\e, \E, in regular expressions 2.7.1
\f (form feed character literal) 2.7.1
\G, in regular expressions 2.7.1
\h, \H, in regular expressions 2.7.1
\k, in regular expressions 2.7.1
\n (newline character literal)
as line feed 2.1.6, 3.3, 4.6, 7.7.2
in regular expressions 2.7.1, 2.7.7
\p, \P, in regular expressions 2.7.1
\Q 2.7.1
\r (carriage return character literal) 2.1.6, 3.3, 7.7.2
in e-mails 4.6
\r, \R, in regular expressions 2.7.1
\s, \S, in regular expressions 2.7.1
\t (tab character literal) 2.7.1
\u (Unicode character literal) 2.7.1
\v, \V, in regular expressions 2.7.1
\w, \W, in regular expressions 2.7.1
\x, in regular expressions 2.7.1
\z, \Z, in regular expressions 2.7.1
^ (caret) 2.7.1, 2.7.7
_ (underscore)
in native method names 13.1
in SQL 5.2, 5.5.3
in URLs 4.3.3
{. . .} (braces)
in glob patterns 2.3.7
in message formatting 7.5.1, 7.5.2

in regular expressions 2.7.1, 2.7.6
| operator
in DTDs 3.4.1
in message formatting 7.5.2
in regular expressions 2.7.1
in text files 2.1.7
~ (tilde), in URLs 4.3.3

A

abort method
of LoginModule 9.4.4
absolute method
of ResultSet 5.6.1, 5.10
AbstractAction class 10.4.5, 11.5.1, 11.8.4
Constructor 11.8.4
AbstractButton class 10.4.7, 11.8.4
isSelected method 11.8.4
setAction method 11.8.4
setActionCommand method 11.8.4
setDisplayedMnemonicIndex method 11.5.5, 11.8.4
setHorizontalTextPosition method 11.5.2, 11.8.4
setMnemonic method 11.8.4
setSelected method 11.8.4
AbstractCellEditor class 12.3.4
isCellEditable method 12.3.4
AbstractTableModel class 12.1.2
getColumnName method 12.1.2
isCellEditable method 12.3.3
Accelerators (in menus) 11.5.5
Accept 4.2.1, 4.2.2
accept method
of FileFilter 11.8.4
of FileFilter (Swing) 11.8.4
of ServerSocket 4.6
acceptChanges method 5.7.2
of CachedRowSet 5.10
Accessory components 11.8.4
Accumulator functions 1.10
accumulator method
of Collector 1.13
Action interface 10.4.5, 10.5, 11.5.1
actionPerformed method 10.4.5
add/removePropertyChangeListener methods 10.4.5
get/putValue methods 10.4.5
getValue method 10.5
is/setEnabled methods 10.4.5
isEnabled method 10.5
predefined action table names in 10.4.5
putValue method 10.5

setEnabled method 10.5
Action listeners 10.4.5
ActionEvent class 10.4.1, 10.4.7
getActionCommand, getModifiers methods 10.4.7
ActionListener interface
actionPerformed method 10.4.1, 10.4.5, 10.4.7
overriding 11.5.1
implementing 10.4.1
ActionMap class 10.4.5
Actions 10.4.5
associating with keystrokes 10.4.5
names of 10.4.5
predefined 10.4.5
Adapter classes 10.4.4
add method
of Area 12.5.4, 12.7.5
of AttributeSet 12.7.5
of ButtonGroup 11.8.4
of Container 10.4.2, 10.5, 11.8.4
of DefaultMutableTreeNode 12.4.1, 12.7.5
of JFrame 10.5
of JMenu 11.5.1, 11.8.4
of JToolBar 11.5.7, 11.8.4
addAttribute method
of AttributesImpl 3.9
addBatch method 5.9.3
of Statement 5.10
addCellEditorListener method
of CellEditor 12.7.5
addChoosableFileFilter method
of JFileChooser 11.8.4
addColumn method 12.2.8
of JTable 12.7.5
addItem method 11.4.4
of JComboBox 11.8.4
addLayoutComponent method
of LayoutManager 11.8.4
addPropertyChangeListener method 10.4.5
addRecipient method 4.6
AddressLayout interface 13.8.3
withTargetLayout method 13.8.3
addSeparator method
of JMenu 11.5.1, 11.8.4
of JToolBar 11.5.7, 11.8.4
addTreeModelListener method 12.4.6
of TreeModel 12.7.5
addTreeSelectionListener method 12.4.5
addURLs method 9.1.2
AdjustmentEvent class 10.4.7
AdjustmentListener interface 10.4.7

Adleman, Leonard 9.3.3, 9.4.4
AES 9.4.1
generating keys in 9.4.2
Affine transformations 12.5.7, 12.6.5
AffineTransform class 12.5.7, 12.7.5
Constructor 12.7.5
getRotateInstance method 12.7.5
getScaleInstance method 12.7.5
getShearInstance method 12.7.5
getTranslateInstance method 12.7.5
getXxxInstance methods 12.5.7, 12.5.8
setToRotation method 12.7.5
setToScale method 12.7.5
setToShear method 12.7.5
setToTranslation method 12.7.5
setToXxx methods 12.5.7
AffineTransformOp class 12.6.5, 12.7.5
Constructor 12.7.5
TYPE_XXX fields 12.6.5
afterLast method 5.6.1
of ResultSet 5.10
Aliases, for namespaces 3.4.2, 3.6
allMatch method 1.6
of Stream 1.13
allocate method
of ByteBuffer 2.4.2, 2.7.7
of SegmentAllocator 13.8.3
allocateFrom method
of SegmentAllocator 13.8.3
Alpha channel 12.5.9
AlphaComposite class 12.5.9, 12.7.5
getInstance method 12.5.9, 12.7.5
Alt+F4 11.5.5
anchor field
of GridBagConstraints 11.8.4
Anchor rectangles 12.5.6
andFilter method 12.2.7
of RowFilter 12.7.5
ANY (DTDs) 3.4.1
anyMatch method 1.6
of Stream 1.13
Apache 3
Derby 5.3
connecting to 5.3.4
drivers for 5.3.2
populating 5.4.4
starting 5.3.3
DOM parser 3.3
Tomcat 5.10
API

Foreign Functions and Memory 13
Apollo 11, launch of 6.2, 6.5
append method
of Appendable 2.1.2, 2.7.7
of JTextArea 11.8.4
of Path2D 12.5.3, 12.5.8, 12.7.5
Appendable interface 2.1.2, 2.7.7
append method 2.7.7
appendChild method 3.8.1
of Node 3.9
Applets
executing 9.3, 9.3.7
security mechanisms in 9
Application servers 5.10
Applications
business logic vs. visual representation of 5.1.2
client/server 5.1.2
closing by user 10.2.1
configuring 3.1
enterprise 5.10
executing
loading classes for 9.1.1
localizing 7.8
paid 9.1.4
server-side 4.3.3
signing 9.3.7
web 5.10
Apply-templates 3.9
applyPattern method
of MessageFormat 7.9
Arc2D class 12.5.2, 12.5.3
Arc2D.Double class 12.5.2, 12.7.5
Constructor 12.7.5
Arc2D.Float class 12.5.2
ArcMaker 12.5.3
Arcs
bounding rectangles of 12.5.2, 12.5.3
closure types of 12.5.3
computing angles of 12.5.3
Area class 12.7.5
add method 12.5.4, 12.7.5
exclusiveOr method 12.5.4, 12.7.5
intersect method 12.5.4, 12.7.5
subtract method 12.5.4, 12.7.5
Arena interface 13.2, 13.3, 13.8.3
automatic 13.3
confined 13.3
global 13.3
ofAuto method 13.8.3
ofConfined method 13.8.3

ofShared method 13.8.3
shared 13.3
ARGB 12.5.9, 12.6.4
ARRAY 5.9.4
Arrays
converting to/from streams 1.2, 1.8, 1.13
getting from a database 5.9.4
of strings 2.7.5
type code for 2.6.2
Arrays class 1.13
stream method 1.2, 1.12, 1.13
Ascender, ascent (in typesetting) 10.3.3
asCharBuffer method
of ByteBuffer 2.7.7
ASCII 2.1.8, 2.7.7
in property files 7.8.2
native code sand 13.1
asMatchPredicate method 2.7.2
asPredicate method 2.7.2
Associative operations 1.10
Atomic operations 2.3.4
ATTLIST (DTDs) 3.4.1
attribute (XML Schema) 3.4.2
Attribute interface 12.7.5
getCategory method 12.7.5
getName method 12.7.5
implementing 12.7.5
Attribute sets 12.7.5
Attributes interface 3.9
enumerating 3.3
getLength method 3.9
getLocalName method 3.9
getQName method 3.9
getURI method 3.9
getValue method 3.9
in XML Schema 3.4.2
legal 3.4.1
names of 3.1
namespace of 3.6
values of 3.1
accessing in XPath 3.5
copying with XSLT 3.9
default (DTDs) 3.4.1
normalizing 3.4.1
vs. elements 3.2, 3.4.1, 3.8.4
AttributeSet interface 12.7.5
add method 12.7.5
get method 12.7.5
remove method 12.7.5
toArray method 12.7.5

AttributesImpl class 3.9
addAttribute method 3.9
clear method 3.9
atZone method 6.5
Authentication 9.2, 9.3.5
role-based 9.2.2
through a trusted intermediary 9.3.5
Autoboxing 12.1.1
AutoCloseable interface
close method 2.1.2
Autocommit mode 5.9.1, 5.9.3
Autoflushing 2.1.6
Automatic arena 13.3
available method
of InputStream 2.1.1, 2.7.7
availableCharsets method 2.1.8
availableLocales method
of Locale 1.9.2
average method
of DoubleStream 1.13
of IntStream 1.13
of LongStream 1.13
average method (primitive streams) 1.12
averaging*Xxx* methods (Collectors) 1.9.4
AWT 10.1, 12.5
event hierarchy in 10.4.7
preferred field sizes in 11.3.1
AWTEvent class 10.4.7

B

B (byte), type code 2.6.2
Bézier curves 12.5.3
Background color
default 10.2.1
setting 10.3.2, 10.4.2
Banding 12.7.1
Banner
getPageCount method 12.7.2
layoutPages method 12.7.2
Banners, printing 12.7.2
Base classes. See Superclasses
Baseline 10.3.3
BaseStream interface 1.13
iterator method 1.13
parallel method 1.13
unordered method 1.13
BasicButtonUI 11.1
BasicFileAttributes interface 2.3.5, 2.7.7
creationTime method 2.7.7

fileKey method 2.7.7
isDirectory method 2.7.7
isOther method 2.7.7
isRegularFile method 2.7.7
isSymbolicLink method 2.7.7
lastAccessTime method 2.7.7
lastModifiedTime method 2.7.7
size method 2.7.7
BasicStroke class 12.5.5, 12.7.5
Constructor 12.7.5
Batch updates 5.9.3
BCP 47 memo 7.1.2
beforeFirst method 5.6.1
of ResultSet 5.10
between method
of Duration 6.1, 6.7
Bevel join 12.5.5
Bicubic, bilinear interpolations 12.6.5
Big-endian order 2.1.8, 2.2.1, 7.7.4
Big5 7.7
BigDecimal class 2.6.4
Binary data
converting to Unicode code units 2.1.4
reading/writing 2.2.1
vs. text 2.1.4
Bindings interface 8.2.2, 8.2.6
get method 8.2.6
put method 8.2.6
Birthdays, calculating 6.2
BitSet class 1.10
BLOB 5.2, 5.9.4
Blob interface 5.5.2, 5.10
getBinaryStream method 5.5.2, 5.10
getBytes method 5.5.2, 5.10
length method 5.10
setBinaryStream method 5.10
BLOBs 5.5.2
placing in database 5.5.2
Blocking
by I/O methods 2.1.1
by network connections 4.1.1, 4.1.3, 4.2.4
Blur filter 12.6.5
BMP 12.6.1
body method
of HttpResponse 4.4.3, 4.6
BodyHandlers
ofString method 4.4.3, 4.4.4
BodyPublishers class
ofString method 4.4.2
Book class 12.7.2

BOOLEAN 5.2, 5.9.4
boolean type
printing 2.1.6
streams of 1.12
type code for 2.6.2
writing in binary format 2.2.1
Bootstrap class loader 9.1.1, 9.1.2
Border layout manager 11.2.2
BorderFactory class 11.8.4
createBevelBorder method 11.8.4
createCompoundBorder method 11.8.4
createEmptyBorder method 11.8.4
createEtchedBorder method 11.8.4
createLineBorder method 11.8.4
createLoweredBevelBorder method 11.8.4
createMatteBorder method 11.8.4
createRaisedBevelBorder method 11.8.4
createTitledBorder method 11.8.4
BorderFactory class, methods of 11.4.3
BorderLayout class 11.2.2, 11.8.4
Constructor 11.8.4
Borders 11.4.3
Bounding rectangle 10.3.1
Bounding rectangles 12.5.2
Box layout 11.6
boxed method
of DoubleStream 1.13
of IntStream 1.13
of LongStream 1.13
boxed method (primitive streams) 1.12
Bray, Tim 3.1
Breadth-first enumerations 12.4.3
breadthFirstEnumeration method 12.4.3
of DefaultMutableTreeNode 12.7.5
Browsers
forms in 4.3.3
response pages in 4.3.3
Buffer class 2.4.2, 2.7.7
clear method 2.4.2, 2.7.7
flip method 2.4.2, 2.7.7
hasRemaining method 2.7.7
limit method 2.7.7
mark method 2.4.2, 2.7.7
remaining method 2.4.2, 2.7.7
reset method 2.4.2, 2.7.7
rewind method 2.4.2, 2.7.7
BufferedImage class 12.5.6, 12.6.4, 12.7.5
constructor 12.6.4, 12.7.5
getColorModel method 12.6.4, 12.7.5
getRaster method 12.6.4, 12.7.5

TYPE_BYTE_GRAY field 12.6.4
TYPE_INT_ARGB field 12.6.4
BufferedImageOp interface 12.6.4, 12.7.5
filter method 12.6.5, 12.7.5
implementing 12.6.5
BufferedInputStream class 2.7.7
Constructor 2.7.7
BufferedOutputStream class 2.7.7
Constructor 2.7.7
Buffers 2.4.2
capacity of 2.4.2
flushing 2.1.1, 2.1.6
in-memory 2.1.2
limits of 2.4.2
marks in 2.4.2
positions in 2.4.1, 2.4.2
traversing all bytes in 2.4.1
vs. random access 2.4.1
build method
of HttpClient.Builder 4.4.1, 4.4.2, 4.6
of HttpRequest.Builder 4.6
Bundle classes 7.8.3
Butt cap 12.5.5
ButtonGroup class 11.4.2, 11.8.4
add method 11.8.4
getSelection method 11.4.2, 11.8.4
ButtonModel interface 11.1, 11.8.4
getActionCommand method 11.4.2, 11.8.4
getSelectedObjects method 11.4.2
properties of 11.1
Buttons
appearance of 11.1
associating actions with 10.4.5
clicking 10.4.2
creating 10.4.2
event handling for 10.4.2
model-view-controller analysis of 11.1
rearranging automatically 11.2.1
ButtonUIListener 11.1
Byte order mark 2.1.8, 7.7.4
byte type
streams of 1.12
type code for 2.6.2
Byte-oriented input/output streams 2.1
ByteArrayClass 8.1.5
ByteArrayClassLoader 8.1.5
ByteArrayOutputStream class 2.6.9
ByteBuffer class 2.4.1, 2.4.2, 2.7.7
allocate method 2.4.2, 2.7.7
asCharBuffer method 2.7.7

compact method 2.7.7
get method 2.7.7
get, get*Xxx* methods 2.4.1
get*Xxx* method 2.7.7
order method 2.4.1, 2.7.7
put method 2.7.7
put, put*Xxx* methods 2.4.1
put*Xxx* method 2.7.7
wrap method 2.4.2, 2.7.7
Bytecodes
engineering
with hex editor 9.1.5
verifying 9.1.5
writing to memory 8.1.5
ByteLookupTable class 12.6.5, 12.7.5
Constructor 12.7.5
byteOffset method
of MemoryLayout 13.8.3
byteOffsetHandle method
of MemoryLayout 13.8.3
Bytes, reading/writing 2.1.1

C

C
calling
from Java programs 13.1
database access in 5.1
FILE* type in 2.1.2
pointers in 13
strings in 13.1
C (char), type code 2.6.2
C++
for native methods 13.1
pointers in 13
Cached row sets 5.7.2, 5.8
CachedRowSet interface 5.7.2, 5.10
acceptChanges method 5.7.2, 5.10
execute method 5.7.2, 5.10
getPageSize method 5.7.2, 5.10
getTableName method 5.7.2, 5.10
nextPage method 5.7.2, 5.10
populate method 5.7.2, 5.10
previousPage method 5.10
setPageSize method 5.7.2, 5.10
setTableName method 5.7.2, 5.10
Caesar cipher 9.1.4
Calendar class 6
formatting objects of 7.3
weekends in 6.2

call escape (SQL) 5.5.3
call method
of CompilationTask 8.1.2
of JavaCompiler.CompilationTask 8.2.6
Callable interface 8.1.2
Callback interface 9.2.2
foreign function 13.7
CallbackHandler interface 9.4.4
handle method 9.4.4
cancelCellEditing method 12.3.4
of CellEditor 12.7.5
cancelRowUpdates method 5.6.2
of ResultSet 5.10
canInsertImage method 12.6.3
of ImageWriter 12.7.5
Canonical layouts 13.5
canonicalLayouts method
of Linker 13.8.3
captureCallState method
of Linker.Option 13.8.3
captureStateLayout method
of Linker.Option 13.8.3
Carriage return character, displaying 3.3
catalog (XML catalogs) 3.4.1
CatalogFeatures class 3.9
defaults method 3.4.1, 3.9
CatalogManager class 3.9
catalogResolver method 3.4.1, 3.9
catalogResolver method
of CatalogManager 3.9
CDATA 3.4.1
CDATA sections 3.2
Cell editors 12.3.3, 12.3.4
Cell renderers
for tables 12.2.1, 12.3.1
for trees 12.4.4
CellEditor interface 12.7.5
addCellEditorListener method 12.7.5
cancelCellEditing method 12.3.4, 12.7.5
getCellEditorValue method 12.3.3, 12.3.4, 12.7.5
isCellEditable method 12.7.5
removeCellEditorListener method 12.7.5
shouldSelectCell method 12.3.4, 12.7.5
stopCellEditing method 12.3.4, 12.7.5
Cells (Swing)
editing 12.3.3
selecting 12.2.5
Certificates 9.2.1, 9.3.4
for software developers 9.3.7
Java Plug-in and 9.3.7

publishing fingerprints of 9.3.4
signing 9.3.6
Chain of trust 9.3.5
ChangeListener interface, stateChanged method 11.4.5
Channels class 4.2.4, 4.6
for files 2.4.1
newInputStream method 4.6
newOutputStream method 4.2.4, 4.6
char type
streams of 1.12
type code for 2.6.2
CHARACTER 5.2, 5.9.4
Character classes 2.7.1
Character encodings 2.1.4, 2.1.8
character order in 7.4
explicitly specified 2.1.8
of source files 7.7.5
partial 2.1.8
platform 2.1.9, 7.7
Character references 3.2
CharacterData interface 3.9
getData method 3.3, 3.9
characteristics method
of Collector 1.13
Characters
differences between 7.4
escaping 2.1.7
normalizing 7.4
outlines of 12.5.8
printing 2.1.6
writing in binary format 2.2.1
characters method
of ContentHandler 3.7.1, 3.9
charAt method
of CharSequence 2.7.7
CharBuffer class 2.1.2, 2.4.2, 2.7.7
get method 2.7.7
put method 2.7.7
CharSequence interface 1.13, 2.1.2, 2.7.7
charAt method 2.7.7
chars method 1.12
codePoints method 1.12, 1.13
length method 2.7.7
splitting 1.2
subSequence method 2.7.7
toString method 2.7.7
Charset class
availableCharsets method 2.1.8
defaultCharset method 2.1.9, 7.7.3
forName method 2.1.8

Checkboxes 11.4.1, 11.5.3, 12.3
checked attribute (HTML) 3.1
checkError method 2.1.6
of PrintWriter 2.7.7
Child elements 3.2
namespace of 3.6
Child nodes 12.4
adding 12.4.1
connecting lines for 12.4.1
children method
of TreeNode 12.7.5
Choice components 11.4
checkboxes 11.4.1, 11.5.3
combo boxes 11.4.4
radio buttons 11.4.2, 11.5.3
sliders 11.4.5
choice element (XML Schema) 3.4.2
choice keyword (message formatting) 7.5.2
Church, Alonzo 6.2
Cipher class 9.4.1, 9.4.4
doFinal method 9.4.1, 9.4.2, 9.4.3, 9.4.4
getBlockSize method 9.4.4
getInstance method 9.4.1, 9.4.4
getOutputSize method 9.4.4
init method 9.4.4
update method 9.4.1, 9.4.2, 9.4.3, 9.4.4
*XXX*_MODE modes 9.4.1
CipherInputStream class 9.4.4
Constructor 9.4.4
read method 9.4.4
CipherOutputStream class 9.4.3, 9.4.4
Constructor 9.4.4
flush method 9.4.4
write method 9.4.4
Ciphers
Caesar 9.1.4
keys for
generating 9.4.2
public 9.3.3, 9.3.5, 9.4.4
performance of 9.4.4
streams for 9.4.3
symmetric 9.4.1, 9.4.4
Class class 9.4.4
forName method 8.1.5
getClassLoader method 9.1.1, 9.4.4
getFields method 12.4.6
Class files 9.1
corrupted 9.1.5
encrypted 9.1.4
loading 9.1.1

portability of 7.7.5
verifying 9.1.5
Class loaders 9.1
as namespaces 9.1.3
bootstrap 9.1.1, 9.1.2
context 9.1.2
extension 9.1.1
hierarchy of 9.1.2
separate for each web page 9.1.3
specifying 9.1.2
system 9.1.1
writing 9.1.4
Class path, adding JAR files to 9.1.2
Classes
adapter 10.4.4
compiling on the fly 8.1.5
externalizable 2.6.2
generic 11.4.4
helper 11.6.8
inheritance trees of 12.4.3
loading 8.1.5, 9.1.1
nonserializable 2.6.3
platform 9.1.1
resolving 9.1.1
separate for each web page 9.1.3
serializable 2.6.1, 2.6.4, 2.6.7
deserializing 2.6.8
static inner 2.6.2
versioning 2.6.7
Classifier functions 1.9.3
ClassLoader class 9.1.1, 9.4.4
defineClass method 9.1.4, 9.4.4
extending 8.1.5, 9.1.4
findClass method 9.1.4, 9.4.4
getParent method 9.4.4
getSystemClassLoader method 9.4.4
loadClass method 9.1.2, 9.1.4
Classloader inversion 9.1.2
CLASSPATH 9.1.1
CLEAR composition rule 12.5.9
clear method
of AttributesImpl 3.9
of Buffer 2.4.2, 2.7.7
clearParameters method
of PreparedStatement 5.10
Client/server applications 5.1.2
Clients
connecting to servers 4.1.2
multiple, serving 4.2.2
clip method

of Graphics2D 12.5.1, 12.5.8, 12.7.1, 12.7.5
Clipping shapes 12.5.1, 12.5.8
printing 12.7.1
setting region for 12.5.1
CLOB data type (SQL) 5.2, 5.9.4
Clob interface 5.5.2, 5.10
getCharacterStream method 5.5.2, 5.10
getSubString method 5.5.2, 5.10
length method 5.10
setCharacterStream method 5.10
CLOBs 5.5.2
placing in database 5.5.2
Clone 2.6.9
clone method
of Object 2.6.1
Cloning 2.6.9
close method
of AutoCloseable 2.1.2
of Closeable 2.1.2, 2.7.7
of Connection 5.4.2, 5.10
of FileLock 2.7.7
of FileSystems 2.7.7
of Flushable 2.1.2
of InputStream 2.1.1, 2.7.7
of OutputStream 2.7.7
of ResultSet 5.4.2, 5.10
of ServerSocket 4.6
of Statement 5.4.2, 5.10
of XMLStreamWriter 3.9
Closeable interface 2.1.2, 2.7.7
close method 2.1.2, 2.7.7
flush method 2.1.2
closeEntry method
of ZipInputStream 2.7.7
of ZipOutputStream 2.7.7
of Zip*Xxx*Stream 2.2.3
closeOnCompletion method 5.4.2
of Statement 5.10
closePath method 12.5.3
of Path2D 12.7.5
Closure types 12.5.3
cmd shell (Windows) 7.7.3
Code units 1.12
in regular expressions 2.7.1
writing 2.2.2
Codebreakers, The (Kahn) 9.1.4
codePoints method 1.12
of CharSequence 1.13
Collation 7.4
CollationKey class 7.9

compareTo method 7.9
Collator class 7.4, 7.9
compare method 7.9
equals method 7.9
getAvailableLocales method 7.9
getCollationKey method 7.4, 7.9
getDecompositon method 7.9
getInstance method 7.9
getStrength method 7.9
setDecomposition method 7.9
setStrength method 7.9
collect method
of Stream 1.9.1, 1.10, 1.13
collectingAndThen method 1.9.4
of Collectors 1.13
Collection interface 1.13
parallelStream method 1.1, 1.13
stream method 1.1, 1.13
Collections class
iterating over elements of 1.1
sort method 7.4
vs. streams 1.1
Collector interface 1.9.1, 1.13
accumulator method 1.13
characteristics method 1.13
combiner method 1.13
finisher method 1.13
of method 1.13
supplier method 1.13
Collectors class 1.9.1, 1.13
averaging*Xxx* methods 1.9.4
collectingAndThen method 1.9.4, 1.13
composing 1.9.4
counting method 1.9.4, 1.13
downstream 1.9.4, 1.13
filtering method 1.9.4, 1.13
flatMapping method 1.9.4, 1.13
groupingBy method 1.9.3, 1.13
groupingByConcurrent method 1.9.3, 1.13
joining method 1.9, 1.13
mapping method 1.9.4, 1.13
maxBy method 1.13
maxBy, minBy methods 1.9.4
minBy method 1.13
partitioningBy method 1.9.3, 1.9.4, 1.13
reducing method 1.9.4
summarizingDouble method 1.13
summarizingInt method 1.13
summarizingLong method 1.13
summarizing*Xxx* methods 1.9, 1.9.4

summingDouble method 1.13
summingInt method 1.13
summingLong method 1.13
summing*Xxx* methods 1.9.4
teeing method 1.9.4
toCollection method 1.9.1, 1.13
toConcurrentMap method 1.9.2, 1.13
toList method 1.9.1, 1.13
toMap method 1.9.2, 1.13
toSet method 1.9.1, 1.9.4, 1.13
toUnmodifiableList method 1.13
toUnmodifiableMap method 1.13
toUnmodifiableSet method 1.13
Color choosers 12.3.4
Color class 10.3.2, 10.5, 12.5.6, 12.7.5
Constructor 10.5, 12.7.5
getRGB method 12.7.5
translating values into pixel data 12.6.4
Color space conversions 12.6.5
ColorConvertOp class 12.6.5
ColorModel class 12.6.4, 12.7.5
getDataElements method 12.7.5
getRGB method 12.6.4, 12.7.5
Colors
background/foreground 10.3.2
changing 10.4.5
components of 12.5.9
composing 12.5.9
interpolating 12.5.6
negating 12.6.5
predefined/custom 10.3.2
solid 12.5.1
Columns 11.3.1
Columns (databases)
accessing by number, in result set 5.4.1
names of 5.2
number of 5.8
Columns (Swing)
accessing 12.2.2
adding 12.2.8
detached 12.1.1
hiding 12.2.8
names of 12.1.2
rendering 12.2.1
resizing 12.1.1, 12.2.3
selecting 12.2.5
com.sun.security.auth.module package 9.2.1
combiner method
of Collector 1.13
Combo box editors 12.3.3

Combo boxes 11.4.4
adding items to 11.4.4
Comments 3.2
commit method
of Connection 5.9.1, 5.10
of LoginModule 9.4.4
commonPool method 1.13
compact method
of ByteBuffer 2.7.7
Comparable interface 1.5
compareTo method 12.2.6
Comparator interface 1.5, 7.4
Comparators class 12.2.6
compare method
of Collator 7.9
compareTo method
of CollationKey 7.9
of Comparable 12.2.6
of Instant 6.1
of String 7.4
Compilable interface 8.2.5, 8.2.6
compile method 8.2.6
CompilationTask 8.1.2
call method 8.1.2
compile method
of Compilable 8.2.6
of Pattern 2.7.2, 2.7.7
CompiledScript class 8.2.6
eval method 8.2.6
Compiler
invoking 8.1.1
Complex types 3.4.2
complexType method 3.4.2
Component class 10.4.7, 10.5, 11.8.4
getBackground method 10.5
getFont method 11.8.4
getForeground method 10.5
getPreferredSize method 10.3, 10.5
getSize method 10.5
inheritance hierarchies of 11.2.1
isVisible method 10.5
repaint method 10.3, 10.5
setBackground method 10.5
setBackground/Foreground methods 10.3.2
setBounds method 10.5
setBounds, setLocation methods 10.2.2
setCursor method 10.5
setForeground method 10.5
setLocation method 10.5
setSize method 10.5

setVisible method 10.2.1, 10.5
validate method 11.8.4
Components (in layout) 11.2.1
classes for 10.2
displaying information in 10.3
labeling 11.3.2
visibility of 10.2.1
Composite interface 12.5.9
Composition rules 12.5.1, 12.5.9
Computer Graphics: Principles and Practice (Hughes et al.) 12.5.3, 12.6.4
concat method
of Stream 1.4, 1.13
Confidential information 9.4
Configuration files 2.5, 2.6.8, 10.5
Confined arena 13.3
Confirmation dialogs 11.8.1
connect method
of Socket 4.6
of URLConnection 4.3.2, 4.6
Connection interface 5.10
close method 5.4.2, 5.10
commit method 5.9.1, 5.10
createBlob method 5.10
createBlob, createClob methods 5.5.2
createClob method 5.10
createStatement method 5.4.1, 5.6.1, 5.6.2, 5.9.1, 5.9.2, 5.10
getAutoCommit method 5.9.3, 5.10
getMetaData method 5.8, 5.10
getWarnings method 5.10
prepareStatement method 5.5.1, 5.6.1, 5.10
releaseSavepoint method 5.9.2, 5.10
rollback method 5.9.1, 5.10
setAutoCommit method 5.9.1, 5.9.3, 5.10
setSavepoint method 5.10
Connections
closing 5.4.2
using row sets after 5.7.2
debugging 4.6
pooling 5.10
starting new threads 4.2.2
console method
of System 7.7.3
Constructive area geometry 12.5.4
Constructors
bypassing 2.6.8
no-argument 2.6.1, 2.6.8
Container class 10.5, 11.2.1, 11.8.4
add method 10.4.2, 10.5, 11.8.4
setLayout method 11.8.4
Containers 11.2.1

Content pane 10.3
Content types 4.3.2
ContentHandler class 3.7.1
characters method 3.7.1
startDocument, endDocument methods 3.7.1
startElement, endElement methods 3.7.1
ContentHandler interface 3.9
characters method 3.9
endDocument method 3.9
endElement method 3.9
startDocument method 3.9
startElement method 3.9
Context class loader 9.1.2
Control points
dragging 12.5.3
of curves 12.5.3
of shapes 12.5.3
Controllers 11.1
convertColumnIndexToModel method
of JTable 12.7.5
convertRowIndexToModel method
of JTable 12.7.5
convertXxxIndexToModel methods
of JTable 12.2.5
Convolution operation 12.6.5
ConvolveOp class 12.6.5, 12.7.5
Constructor 12.7.5
CookieHandler class
setDefault method 4.3.3
Cookies 4.3.3
Coordinate system
custom 12.5.1
translating 12.7.1
Coordinate transformations 12.5.7
Copies 12.7.5
getValue method 12.7.5
CopiesSupported 12.7.5
copy method
of Files 2.3.4, 2.3.9, 2.7.7
copyArea method
of Graphics 10.5
CORBA 9.1.1
Core Swing (Topley) 12.1, 12.4, 12.4.2
count function
of XPath 3.5
count method
of Stream 1.1, 1.6, 1.13
counting method
of Collectors 1.9.4, 1.13
Country codes 1.9.3, 7.1.2

CRC32 checksum 2.4.1
CRC32 class 2.4.1
CREATE TABLE 5.2
executing 5.4.1
in batch updates 5.9.3
createBevelBorder method
of BorderFactory 11.8.4
createBindings method
of ScriptEngine 8.2.6
createBlob method
of Connection 5.10
createBlob, createClob methods
of Connection 5.5.2
createCachedRowSet method 5.7.1, 5.7.2
of RowSetFactory 5.10
createClob method
of Connection 5.10
createCompoundBorder method
of BorderFactory 11.8.4
createDirectories method
of Files 2.7.7
createDirectory method
of Files 2.7.7
createDirectory, createDirectories methods
of Files 2.3.3
createElement method
of Document 3.9
createElement, createElementNS methods
of Document 3.8.1
createElementNS method
of Document 3.9
createEmptyBorder method
of BorderFactory 11.8.4
createEtchedBorder method
of BorderFactory 11.8.4
createFile method 2.3.3
of Files 2.7.7
createFilteredRowSet method
of RowSetFactory 5.10
createImageInputStream method
of ImageIO 12.7.5
createImageOutputStream method
of ImageIO 12.7.5
createImageXxxStream methods of ImageIO
of ImageIO 12.6.3
createJdbcRowSet method
of RowSetFactory 5.10
createJoinRowSet method
of RowSetFactory 5.10
createLineBorder method

of BorderFactory 11.8.4
createLink method
of Files 2.7.7
createLoweredBevelBorder method
of BorderFactory 11.8.4
createLSOutput method 3.8.3
createLSSerializer method 3.8.3
createMatteBorder method
of BorderFactory 11.8.4
createPrintJob method 12.7.3
of PrintService 12.7.5
createRaisedBevelBorder method
of BorderFactory 11.8.4
createServerSocket method
of ServerSocketFactory 4.6
createSocket method
of SocketFactory 4.6
createStatement method 5.4.1, 5.6.1, 5.6.2, 5.9.1, 5.9.2
of Connection 5.10
createSymbolicLink method
of Files 2.7.7
createTempDirectory method
of Files 2.7.7
createTempFile method
of Files 2.7.7
createTemp*Xxx* methods
of Files 2.3.3
createTextNode method 3.8.1
of Document 3.9
createTitledBorder method
of BorderFactory 11.8.4
createTypeBorder methods (BorderFactory) 11.4.3
createWebRowSet method
of RowSetFactory 5.10
createXMLStreamReader method
of XMLInputFactory 3.9
createXMLStreamWriter method 3.8.4
of XMLOutputFactory 3.9
creationTime method
of BasicFileAttributes 2.7.7
Credit card numbers 9.4
Cryptography and Network Security (Stallings) 9.3.1
Ctrl+O, Ctrl+S accelerators 11.5.5
Cubic curves 12.5.3
CubicCurve2D class 12.5.2, 12.5.3
CubicCurve2D.Double class 12.5.2, 12.7.5
Constructor 12.7.5
CubicCurve2D.Float class 12.5.2
Currencies 7.2.3
formatting 7.2.2

identifiers for 7.2.3
Currency class 7.2.3, 7.9
getAvailableCurrencies method 7.9
getCurrencyCode method 7.9
getDefaultFractionDigits method 7.9
getInstance method 7.2.3, 7.9
getNumericCode method 7.9
getNumericCodeAsString method 7.9
getSymbol method 7.9
toString method 7.9
Current user 10.5
Cursor class, getPredefinedCursor method 10.4.6
Cursor shapes 10.4.6
curveTo method 12.5.3
of Path2D.Float 12.7.5
Custom layout managers 11.7
Customizations. See Preferences
Cyclic references 2.6.8

D

D (double), type code 2.6.2
d (SQL escape) 5.5.3
Dashed lines 12.5.5
Data
encrypting/decrypting 9.4.3
fingerprints of 9.3.1
signed 9.3.3
Data exchange 11.8.3
Data sources (for JNDI service) 5.10
Data types
codes for 2.6.2
print services for 12.7.3
database.properties package 5.4.4, 5.10
DatabaseMetaData interface 5.8, 5.10
getJDBCMajorVersion method 5.10
getJDBCMinorVersion method 5.10
getMaxConnections method 5.10
getMaxStatements method 5.4.2, 5.10
getSQLStateType method 5.4.3
getTables method 5.8, 5.10
supportsBatchUpdates method 5.9.3, 5.10
supportsResultSetConcurrency method 5.10
supportsResultSetType method 5.10
supportsResultSet*Xxx* methods 5.6.1
Databases 5, 5.10
accessing, in C language 5.1
autocommit mode of 5.9.1, 5.9.3
batch updates for 5.9.3
caching prepared statements 5.5.1

changing data with SQL 5.2
connections to 5.3.1, 5.3.4, 5.4.4
closing 5.4.2, 5.4.4
in web and enterprise applications 5.10
pooling 5.10
drivers for 5.1.1
duplicating data in 5.2
error handling in 5.9.3
escape syntax in 5.5.3
integrity of 5.9
JAR files for 5.3.2
keys in 5.5.5
LOBs in 5.5.2
metadata for 5.8
modifying 5.7.2
native storage for XML in 5.9.4
numbering columns in 5.4.1
outer joins in 5.5.3
populating 5.4.4
registering classes for 5.3.4
scalar functions in 5.5.3
scrollable/updatable result sets in 5.6.1
setting up parameters in 5.7.2
starting 5.3.3
stored procedures in 5.5.3
structure of 5.2, 5.8
synchronization of 5.7.2
tools for 5.8
truncated data from 5.4.3
URLs of 5.3.1
DataInput interface 2.7.7
readBoolean method 2.2.1, 2.7.7
readByte method 2.7.7
readChar method 2.2.1, 2.2.2, 2.7.7
readDouble method 2.2.1, 2.6.1, 2.6.4, 2.7.7
readFloat method 2.2.1, 2.7.7
readFully method 2.7.7
readInt method 2.2.1, 2.2.2, 2.6.1, 2.7.7
readLong method 2.2.1, 2.7.7
readShort method 2.2.1, 2.7.7
readUTF method 2.2.1, 2.7.7
skipBytes method 2.7.7
DataInputStream class 2.1.2, 2.1.3, 2.2.1
DataIO
*xxx*FixedString methods 2.2.2
DataOutput interface 2.2.1, 2.7.7
writeBoolean method 2.2.1, 2.7.7
writeByte method 2.2.1, 2.7.7
writeChar method 2.2.1, 2.2.2, 2.7.7
writeChars method 2.2.1, 2.7.7

writeDouble method 2.2.1, 2.6.1, 2.6.4, 2.7.7
writeFloat method 2.2.1, 2.7.7
writeInt method 2.2.1, 2.2.2, 2.6.1, 2.7.7
writeLong method 2.2.1, 2.7.7
writeShort method 2.2.1, 2.7.7
writeUTF method 2.2.1, 2.7.7
DataOutputStream class 2.1.2, 2.2.1
DataSource interface 5.10
DataTruncation class 5.4.3, 5.10
getDataSize method 5.10
getIndex method 5.10
getParameter method 5.10
getTransferSize method 5.10
DATE 5.2, 5.5.3, 5.9.4
Date and Time API 6, 6.7
legacy code and 6.7
Date class 2.6.2, 2.6.4, 6, 6.7
formatting objects of 7.3
toInstant method 6.7
valueOf method 6.7
Date class
months and years in 6.2
dateFilter method 12.2.7
of RowFilter 12.7.5
DateFormat class 6.6, 6.7, 7.3
DateFormatter class 7.3
Dates
computing 6.2, 6.3
filtering 12.2.7
formatting 6.6, 7.1.1, 7.3, 7.5.1
literals for 5.5.3
local 6.2
nonexistent 7.3
parsing 6.6
datesUntil method 6.2
of LocalDate 6.7
DateTimeFormatter class 6.6, 6.7, 7.3, 7.9
format method 6.6, 6.7, 7.9
legacy classes and 6.7
ofLocalizedDate method 6.7, 7.9
ofLocalizedDateTime method 6.7, 7.9
ofLocalizedPattern method 6.7
ofLocalizedTime method 6.7, 7.9
ofLocalizedXxx methods 6.6, 7.3
ofPattern method 6.6, 6.7
parse method 6.6
toFormat method 6.6, 6.7
withLocale method 6.6, 6.7, 7.3, 7.9
DateTimeSyntax class 12.7.5
Daylight savings time 6.5

DayOfWeek enumeration 6.2
getDisplayName method 6.6, 7.3
dayOfWeekInMonth method
of TemporalAdjusters 6.7
Days of week 7.3
DBeaver 5.8
Debugging
intermittent bugs 10.2.1
JDBC-related problems 5.3.4
mail connections 4.6
streams 1.5
with telnet 4.1.1
DECIMAL 5.2, 5.9.4
Decimal separators 7.1.1, 7.2.1, 7.2.2
DecimalFormat class 7.2.2
DecimalFormatSymbols class 7.2.2
decode method
of URLDecoder 4.6
Decomposition 7.4
DefaultButtonModel class 11.1
DefaultCellEditor class 12.4.2, 12.7.5
Constructor 12.7.5
variations of 12.3.3
defaultCharset method 2.1.9, 7.7.3
DefaultComboBoxModel class 11.4.4
DefaultHandler class 3.7.1
defaultLookup method
of Linker 13.8.3
DefaultMutableTreeNode class 12.4.1, 12.4.3, 12.4.4, 12.7.5
add method 12.4.1, 12.7.5
breadthFirstEnumeration method 12.7.5
Constructor 12.7.5
depthFirstEnumeration method 12.7.5
pathFromAncestorEnumeration method 12.4.3
postOrderEnumeration method 12.7.5
preOrderEnumeration method 12.7.5
setAllowsChildren method 12.4.1, 12.7.5
xxxFirstEnumeration methods 12.4.3
xxxOrderEnumeration methods 12.4.3
defaultPage method
of PrinterJob 12.7.5
DefaultRowSorter class 12.7.5
setComparator method 12.2.6, 12.7.5
setRowFilter method 12.2.7, 12.7.5
setSortable method 12.2.6, 12.7.5
defaults method
of CatalogFeatures 3.4.1, 3.9
DefaultTableCellRenderer class 12.3.1
DefaultTableModel class
isCellEditable method 12.3.3

DefaultTreeCellRenderer class 12.4.4, 12.7.5
getTreeCellRendererComponent method 12.4.4
setClosedIcon method 12.7.5
setLeafIcon method 12.7.5
setOpenIcon method 12.7.5
setXxxIcon methods 12.4.4
DefaultTreeModel class 12.4.1, 12.4.2, 12.4.6, 12.7.5
automatic notifications by 12.4.2
getPathToRoot method 12.4.2
insertNodeInto method 12.4.2, 12.7.5
isLeaf method 12.4.1
nodeChanged method 12.4.2, 12.7.5
nodesChanged method 12.7.5
reload method 12.4.2, 12.7.5
removeNodeFromParent method 12.4.2, 12.7.5
setAsksAllowsChildren method 12.4.1, 12.7.5
defaultWriteObject method 2.6.4
defineClass method 9.1.4
of ClassLoader 9.4.4
DELETE 5.2
executing 5.4.1
in batch updates 5.9.3
vs. methods of ResultSet 5.6.2
delete method
of Files 2.3.4, 2.7.7
of HttpRequest.Builder 4.6
deleteIfExists method 2.3.4
of Files 2.7.7
deleteRow method 5.6.2
of ResultSet 5.10
Delimiters, in text files 2.1.7
Depth-first enumerations 12.4.3
depthFirstEnumeration method 12.4.3
of DefaultMutableTreeNode 12.7.5
derbyclient.jar package 5.3.2
dereferenceElement method
of MemoryLayout.PathElement 13.8.3
DES 9.4.1
Descender, descent (in typesetting) 10.3.3
Deserialization 2.6.8
Device coordinates 12.5.7
Diagnostic interface 8.1.3, 8.2.6
getColumnNumber method 8.2.6
getKind method 8.2.6
getLineNumber method 8.2.6
getMessage method 8.2.6
getSource method 8.2.6
DiagnosticCollector class 8.1.3, 8.2.6
Constructor 8.2.6
getDiagnostics method 8.2.6

DiagnosticListener interface 8.1.3
Dialogs 11.8
accepting/canceling 11.8.3
closing 10.4.4, 11.5.5, 11.8.2, 11.8.3
confirmation 11.8.1
creating 11.8.2
data exchange in 11.8.3
default button in 11.8.3
displaying 11.8.2
modal 11.8
modeless 11.8, 11.8.2, 11.8.3
root pane of 11.8.3
Dictionary ordering 7.4
digest method
of MessageDigest 9.3.1, 9.4.4
DigiCert 9.3.4, 9.3.5
Digital fingerprints 2.6.2, 9.3.1
computing 2.6.7
different for a class and its objects 2.6.2
Digital signatures 9.3
verifying 9.3.4
Directories
creating 2.3.3
current 2.3.8
hierarchical structure of 12.4
printing all subdirectories of 2.3.8
traversing 2.3.6
user's working 2.1.3
DirectoryStream interface 2.3.8
discarding method
of HttpResponse.BodyHandlers 4.4.3
distinct method
of Stream 1.5, 1.13
dividedBy method
of Duration 6.7
Do-nothing methods 10.4.4
Doc interface 12.7.3
DocAttribute interface 12.7.5
implementing 12.7.5
printing attributes of 12.7.5
DocAttributeSet interface 12.7.5
DocFlavor class 12.7.3, 12.7.4
DocPrintJob interface 12.7.5
getAttributes method 12.7.5
print method 12.7.5
DOCTYPE 3.4.1
including in output 3.8.3
Document flavors, for print services 12.7.3
Document interface 3.3, 3.9
createElement method 3.9

createElementNS method 3.9
createTextNode method 3.9
createXxx methods 3.8.1
getDocumentElement method 3.3, 3.9
DocumentBuilder class 3.9
newDocument method 3.8.1, 3.9
parse method 3.9
setEntityResolver method 3.4.1, 3.9
setErrorHandler method 3.9
DocumentBuilderFactory class 3.9
isIgnoringElementContentWhitespace method 3.9
isNamespaceAware method 3.9
isValidating method 3.9
newDocumentBuilder method 3.3, 3.8.2, 3.9
newInstance method 3.3, 3.6, 3.9
newNSInstance method 3.6, 3.7.1, 3.9
setIgnoringElementContentWhitespace method 3.4.1, 3.9
setNamespaceAware method 3.4.2, 3.6, 3.7.1, 3.8.2, 3.9
setValidating method 3.4.1, 3.9
doFinal method 9.4.1, 9.4.2, 9.4.3
of Cipher 9.4.4
DOM (Document Object Model) parser 3.3, 3.7
namespace-awareness of 3.6, 3.7.1
trees in
accessing with XPath 3.5
analyzing 3.3
building 3.7.1, 3.8, 3.9
writing 3.8.3
DOMImplementationLS
createLSOutput method 3.8.3
createLSSerializer method 3.8.3
DOMResult class 3.9
Constructor 3.9
DOMSource class 3.9
Constructor 3.9
DOUBLE 5.2, 5.9.4
double type
printing 2.1.6
streams of 1.12
type code for 2.6.2
writing in binary format 2.2.1
DoubleBuffer class 2.4.2
doubles method
of RandomGenerator 1.12, 1.13
DoubleStream interface 1.12, 1.13
average method 1.12, 1.13
boxed method 1.12, 1.13
mapToDouble method 1.12
max method 1.13
max, min methods 1.12

min method 1.13
of method 1.13
sum method 1.13
sum, summaryStatistics methods 1.12
summaryStatistics method 1.13
toArray method 1.12, 1.13
DoubleSummaryStatistics class 1.9, 1.12, 1.13
getAverage method 1.13
getCount method 1.13
getMax method 1.13
getMin method 1.13
getSum method 1.13
doubleValue method 7.2.1
Downcall handle 13.6
downcallHandle method
of Linker 13.8.3
Downstream collectors 1.9.4, 1.13
draw method
of Graphics2D 12.5.1, 12.5.2, 12.5.4, 12.5.5, 12.7.5
of Shape 10.3.1
drawImage method 10.3.4
of Graphics 10.5
Drawing with mouse 10.4.6
Drawings
creating 12.5
printing 12.7.1
drawString method
of Graphics2D 10.5
drawXxx methods
of Graphics 12.5.2
DriverManager class 5.10
getConnection method 5.3.4, 5.4.4, 5.10
setLogWriter method 5.3.4
DROP TABLE 5.3.4
executing 5.4.1
in batch updates 5.9.3
Drop-down lists 11.4.4
dropWhile method 1.4
of Stream 1.13
DSA 9.3.3
DST, DST_Xxx composition rules 12.5.9
DTDHandler interface 3.7.1
DTDs 3.4
element content in 3.4.1
entities in 3.4.1
external 3.4.1
in XML documents 3.2, 3.4.1
locating 3.4.1
unambiguous 3.4.1
URLs for 3.4.1

Duration class 6.7
between method 6.1, 6.7
dividedBy method 6.7
getSeconds method 6.1
immutability of 6.1
isNegative method 6.7
isZero method 6.7
minus method 6.7
minus*Xxx* method 6.7
multipliedBy method 6.7
negated method 6.7
of*Xxx* method 6.7
plus method 6.7
plus*Xxx* method 6.7
toDaysPart method 6.7
toSeconds method 6.7
to*Xxx* method 6.7
toXxx methods 6.1
to*Xxx*Part method 6.7
Dynamic web pages 8.1.6

E

E-mails
sending 4.6
terminating lines in 4.6
Echo character 11.3.3
Echo servers 4.2.1
Edge detection 12.6.5
ELEMENT (DTDs) 3.4.1
element (XML Schema) 3.4.2
Element interface 3.9
getAttribute method 3.3, 3.9
getTagName method 3.3, 3.6, 3.9
of Document 3.3
setAttribute method 3.9
setAttribute, setAttributeNS methods 3.8.1
setAttributeNS method 3.9
Elements
child 3.2
accessing in XPath 3.5
namespace of 3.6
constructing 3.8.1
counting, in XPath 3.5
empty 3.1
legal attributes of 3.4.1
mixed content in 3.2, 3.4.1
names of 3.3, 3.6
root 3.2, 3.4.2
stripping whitespace in 3.3

vs. attributes 3.2, 3.4.1, 3.8.4
elements method
of SegmentAllocator 13.8.3
Ellipse2D class 10.3.1, 12.5.2
setFrameFromCenter method 10.3.1
Ellipse2D.Double class 10.5
Constructor 10.5
Ellipse2D.Double, Ellipse2D.Float classes 12.5.2
Ellipses 10.3.1, 12.5.2
bounding rectangles of 10.3.1
constructing 10.3.1
filling with color 10.3.2
EMPTY (DTDs) 3.4.1
empty method
of Optional 1.7.5, 1.13
of Stream 1.2, 1.13
Empty tags 3.1
encode method
of URLEncoder 4.6
Encodings. See Character encodings
Encryption 9.4
decryption keys for 9.1.4
exporting strong methods of 9.1.4
final block padding in 9.4.1
of class files 9.1.4
End cap styles 12.5.5
end method
of MatchResult 2.7.7
End points 12.5.3
End tags 3.1
End-of-line character. See Line feed
endDocument method 3.7.1
of ContentHandler 3.9
endElement method 3.7.1
of ContentHandler 3.9
Enterprise applications 5.10
Enterprise JavaBeans 5.1.2
Entity references 3.2, 3.4.1
Entity resolvers 3.3, 3.4.1
ENTITY, ENTITIES attribute types (DTDs) 3.4.1
EntityResolver interface 3.7.1, 3.9
resolveEntity method 3.4.1, 3.9
Entrust 9.3.5
Entry 12.2.7
enumeration (XML Schema) 3.4.2
Enumeration interface
nextElement method 12.4.3
Enumerations
ignoring serialVersionUID fields 2.6.7
of nodes, in a tree 12.4.3

serialization of 2.6.6
using attributes for 3.4.1
EnumSyntax class 12.7.5
Epoch 2.6.4, 6.1
equals method
of Collator 7.9
of Instant 6.1
equiDoubles method
of RandomGenerator 1.13
errno
capturing 13.8.3
Error 3.4.1
Error handlers
installing 3.4.1
error method
of ErrorHandler 3.9
ErrorHandler interface 3.7.1, 3.9
error method 3.4.1, 3.9
fatalError method 3.4.1, 3.9
warning method 3.4.1, 3.9
Errors
UnsatisfiedLinkError 13.1
Escapes
in regular expressions 2.1.7
in SQL 5.5.3
Essential XML (Box et al.) 3, 3.9
Euro symbol 7.2.3, 7.7.3
eval method
of CompiledScript 8.2.6
of ScriptEngine 8.2.2, 8.2.6
evaluate method
of XPath 3.9
evaluate, evaluateExpression methods (XPath) 3.5
evaluateExpression method
of XPath 3.9
Event delegation model 10.4.1
Event dispatch thread 10.2.1
Event handling 10.4, 10.4.7
semantic vs. low-level events 10.4.7
Event listeners 10.4.1
with lambda expressions 10.4.3
Event objects 10.4.1
Event procedures 10.4
Event sources 10.4.1
EventListenerList class 12.4.6
EventObject class 10.4.1
getActionCommand, getSource methods 10.4.7
Evins, Jim 12.2.1
Exceptions
FileNotFoundException 4.3.3

IllegalStateException 1.9.2, 12.6.3
in SQL 5.4.3
IndexOutOfBoundsException 12.6.3
InvalidObjectException 2.6.7
InvalidPathException 2.3.1
IOException 2.1.2, 4.1.2
MissingResourceException 7.8.1
NotSerializableException 2.6.4
OverlappingFileLockException 2.5
ParseException 7.2.1
ReadOnlyBufferException 2.4.1
SocketTimeoutException 4.1.3
SQLException 5.4.3, 5.6.1, 5.9.1
SQLWarning 5.6.1
SyncProviderException 5.7.2
UnknownHostException 4.1.2
Exclusive lock 2.5
exclusiveOr method 12.5.4
of Area 12.7.5
exec method
of Runtime 2.6.8
execute method
of CachedRowSet 5.7.2, 5.10
of RowSet 5.10
of Statement 5.4.4, 5.5.4, 5.10
executeBatch method 5.9.3
of Statement 5.10
executeLargeBatch method
of Statement 5.10
executeLargeUpdate method
of Statement 5.10
executeQuery method
of PreparedStatement 5.5.1, 5.10
of Statement 5.4.1, 5.6.1, 5.6.2, 5.10
executeUpdate method
of PreparedStatement 5.5.1, 5.10
of Statement 5.4.1, 5.9.1, 5.9.2, 5.10
executor method
of HttpClient.Builder 4.6
ExecutorService interface 8.1.2
exists method
of Files 2.3.5, 2.7.7
EXIT 5.3.3
exportNode method
of Preferences 10.5
exportSubtree method
of Preferences 10.5
export*Xxx* methods
of Preferences 10.5
Extension class loader 9.1.1

extern "C", in native methods (C++) 13.1
External entities 3.4.1
External padding 11.6.5
Externalizable interface
methods of 2.6.5

F

F (float), type code 2.6.2
Factoring algorithms 9.3.3
fatalError method 3.4.1
of ErrorHandler 3.9
Field class
getName, getType methods 12.4.6
Fields
of deserialized objects 2.6.6
transient 2.6.3, 2.6.4
File class 2.7.7
separator constant 2.1.3
toPath method 2.3.1, 2.7.7
File dialogs 11.8.4
adding accessory components to 11.8.4
File pointers 2.2.2
File systems, POSIX-compliant 2.3.5
file.encoding package 2.1.9
file: scheme 4.3.1
FileChannel class 2.7.7
lock method 2.5, 2.7.7
map method 2.4.1, 2.7.7
open method 2.4.1, 2.7.7
truncate method 2.7.7
tryLock method 2.5, 2.7.7
FileFilter interface 11.8.4
FileFilter class 11.8.4
accept method 11.8.4
getDescription method 11.8.4
FileFilter class (Swing), methods of 11.8.4
FileInputStream class 2.1.3, 2.7.7
Constructor 2.7.7
getChannel method 2.7.7
read method 2.1.1
fileKey method
of BasicFileAttributes 2.7.7
FileLock class 2.7.7
close method 2.7.7
isShared method 2.5
FileNameExtensionFilter class 11.8.4
Constructor 11.8.4
FileNotFoundException class 4.3.3
FileOutputStream class 2.1.3, 2.7.7

Constructor 2.7.7
getChannel method 2.7.7
Files class 1.13, 2.3, 2.3.2, 2.7.7
channels for 2.4.1
closing 2.3.6, 2.3.8
configuration 2.5
copy method 2.3.4, 2.3.9, 2.7.7
copying 2.3.4
createDirectories method 2.7.7
createDirectory method 2.7.7
createFile method 2.7.7
createLink method 2.7.7
createSymbolicLink method 2.7.7
createTempDirectory method 2.7.7
createTempFile method 2.7.7
create*Xxx* methods 2.3.3
creating 2.3.3
delete method 2.7.7
delete, deleteIfExists methods 2.3.4
deleteIfExists method 2.7.7
deleting 2.3.4
encrypted data in 9.4.3
exists method 2.3.5, 2.7.7
filtering 2.3.7, 2.3.8, 12.6.2
filters for 11.8.4
find method 2.3.6
getOwner method 2.3.5
hierarchical structure of 12.4
isDirectory method 2.7.7
isExecutable method 2.7.7
isHidden method 2.7.7
isReadable method 2.7.7
isRegularFile method 2.7.7
isSymbolicLink method 2.7.7
isWritable method 2.7.7
is*Xxx* methods 2.3.5
lines method 1.2, 1.13, 2.3.2
list method 2.3.6
locking 2.5
memory-mapped 1.13, 2.4
MIME type of 2.3.2
mismatch method 2.3.2, 2.7.7
missing 8.1.3
move method 2.3.4, 2.7.7
moving 2.3.4
newBufferedReader method 2.7.7
newBufferedWriter method 2.7.7
newBuffered*Xxx* methods 2.3.2
newDirectoryStream method 2.3.8, 2.7.7
newInputStream method 2.7.7

newOutputStream method 2.7.7
new*Xxx*Stream methods 2.3.2
opening/saving in GUI 11.8.4
probeContentType method 2.3.2, 2.7.7
random-access 2.2.2, 2.4.1
readAllBytes method 2.7.7
readAllLines method 2.7.7
readAll*Xxx* methods 2.3.2
readAttributes method 2.3.5, 2.7.7
reading/writing 2.1.3, 2.3.2
by one byte 2.1.1
readString method 2.3.2, 2.7.7
size method 2.3.5, 2.7.7
total number of bytes in 2.2.2
traversing 2.3.8
walk method 2.3.6
walkFileTree method 2.3.8, 2.7.7
with multiple images 12.6.3
write method 2.7.7
write, writeString methods 2.3.2
writeString method 2.7.7
FileSystem class 2.7.7
getPath method 2.3.9, 2.7.7
getPathMatcher method 2.7.7
FileSystems class 2.7.7
close method 2.7.7
getDefault method 2.7.7
newFileSystem method 2.3.9, 2.7.7
FileTime class 2.7.7
toInstant method 2.7.7, 6.7
FileTypeDetector 2.3.2
FileView class 11.8.4
getDescription method 11.8.4
getIcon method 11.8.4
getName method 11.8.4
getTypeDescription method 11.8.4
isTraversable method 11.8.4
FileView class, methods of 11.8.4
FileVisitor interface 2.3.8
methods of 2.3.8
Fill 12.5.1, 12.5.4
fill field
of GridBagConstraints 11.8.4
fill method
of Graphics2D 10.3.2, 10.5, 12.7.5
Filling shapes 12.5.1
fillXxx methods
of Graphics 12.5.2
filter method
of BufferedImageOp 12.6.5, 12.7.5

of Optional 1.7.3, 1.13
of Stream 1.1, 1.3, 1.6, 1.13
FilteredRowSet interface 5.7.1
filtering method
of Collectors 1.9.4, 1.13
Filters
for images 12.6.5
for numbers 12.2.7
for table rows 12.2.7
glob patterns for 2.3.7, 2.3.8
implementing 12.2.7
Filter*Xxx*Stream classes 2.1.3
Final block padding 9.4.1
find method
of Files 2.3.6
of Matcher 2.7.7
of SymbolLookup 13.8.3
findAll method
of Scanner 2.7.7
findAny method 1.6
of Stream 1.13
findClass method 9.1.4
of ClassLoader 9.4.4
findColumn method
of ResultSet 5.10
findFirst method 1.6
of Stream 1.13
findOrThrow method
of SymbolLookup 13.8.3
findStatic method
of MethodHandle.Lookup 13.8.3
Fingerprints. See Digital fingerprints
finisher method
of Collector 1.13
first method
of ResultSet 5.6.1, 5.10
firstDayOfMonth method
of TemporalAdjusters 6.7
firstDayOfNextMonth method
of TemporalAdjusters 6.7
firstDayOfNextYear method
of TemporalAdjusters 6.7
firstDayOfYear method
of TemporalAdjusters 6.7
firstValue method 4.4.3
of HttpHeaders 4.6
firstVariadicArg method
of Linker.Option 13.8.3
#FIXED 3.4.1
Flash 10.1

flatMap method
of Optional 1.7.6, 1.13
of Stream 1.3, 1.13
flatMapping method 1.9.4
of Collectors 1.13
flip method
of Buffer 2.4.2, 2.7.7
FLOAT 5.2, 5.9.4
float type
printing 2.1.6
streams of 1.12
type code for 2.6.2
writing in binary format 2.2.1
FloatBuffer class 2.4.2
Floating-point numbers 7.1.1, 7.2
Flow layout manager 11.2.1
FlowLayout class 11.8.4
Constructor 11.8.4
flush method
of CipherOutputStream 9.4.4
of Closeable 2.1.2
of Flushable 2.1.2, 2.7.7
of OutputStream 2.1.1, 2.7.7
Flushable interface 2.1.2, 2.7.7
close method 2.1.2
flush method 2.1.2, 2.7.7
fn (SQL escape) 5.5.3
FocusEvent class 10.4.7
isTemporary method 10.4.7
FocusListener interface, methods of 10.4.7
fold method
of Gatherers 1.13
Folders, icons for 12.4.1, 12.4.4
followRedirects method 4.4.1
of HttpClient.Builder 4.6
Font class 10.3.3, 10.5
Constructor 10.5
getFamily method 10.5
getFontName method 10.5
getLineMetrics method 10.3.3, 10.5
getName method 10.5
getStringBounds method 10.3.3, 10.5
Font render context 12.5.8
FontMetrics class 10.5
getFontRenderContext method 10.5
Fonts 10.3.3
checking availability of 10.3.3
names of 10.3.3
size of 10.3.3
styles of 10.3.3

typesetting properties of 10.3.3
forEach method 1.8
of Stream 1.13
forEachOrdered method 1.8
Foreground color 10.3.2
Foreign function
callback 13.7
Foreign functions 13.2
Foreign Functions and Memory API 13
Forest 12.4, 12.4.1
ForkJoinPool class
commonPool method 1.13
forLanguageTag method 7.1.2
of Locale 7.9
Format class 6.7, 7.9
format method 7.9
format method
of DateTimeFormatter 6.6, 6.7, 7.9
of Format 7.9
of LocalDate 7.3
of LocalDateTime 7.3
of LocalTime 7.3
of MessageFormat 7.5.1, 7.9
of NumberFormat 7.2.1, 7.9
of String 7.1.4
of ZonedDateTime 7.3
Formatting
dates 7.1.1, 7.3, 7.5.1
messages 7.5
numbers 7.1.1, 7.2, 7.5.1
Forms, processing 4.3.3
forName method
of Charset 2.1.8
of Class 8.1.5
ForwardingJavaFileManager class 8.2.6
ForwardingJavaFileManager method 8.2.6
getJavaFileForOutput method 8.2.6
ForwardingJavaFileManager method
of ForwardingJavaFileManager 8.2.6
Frame class 10.2, 10.5
getIconImage method 10.5
getTitle method 10.5
isResizable method 10.5
setIconImage method 10.2.2, 10.5
setResizable method 10.2.2, 10.5
setTitle method 10.2.2, 10.5
Frames
closing by user 10.2.1
creating 10.2
displaying information in 10.3

positioning 10.2.2
properties of 10.2.2
FROM 5.2
from method
of Instant 6.7
of ZonedDateTime 6.7
FTP 4.3.2
ftp: scheme 4.3.1, 4.3.2
Function interface
identity method 1.9.2
FunctionDescriptor interface 13.8.3
of method 13.8.3
ofVoid method 13.8.3
Functions. See Methods

G

Gödel's theorem 9.1.5
Gadget chains 2.6.8
Garbage collection
native methods and 13.1
gather method
of Stream 1.13
Gatherer interface 1.13
initializer method 1.13
of method 1.13
ofSequential method 1.13
Gatherer.Downstream interface 1.13
isRejecting method 1.13
push method 1.13
Gatherer.Integrator interface 1.13
integrate method 1.13
of method 1.13
Gatherers class 1.13
fold method 1.13
mapConcurrent method 1.13
windowFixed method 1.13
GeneralPath class 12.5.2, 12.5.3, 12.7.5
Constructor 12.7.5
Generate 1.12
generate method
of Stream 1.2, 1.13
generateKey method 9.4.2
of KeyGenerator 9.4.4
Generators, converting to streams 1.13
Generic programming
classes in 11.4.4
GET 4.3.3
building 4.4.2
get method

of AttributeSet 12.7.5
of Bindings 8.2.6
of ByteBuffer 2.4.1, 2.7.7
of CharBuffer 2.7.7
of HttpRequest.Builder 4.4.2, 4.6
of MemorySegment 13.8.3
of Optional 1.7.4, 1.7.7, 1.13
of Paths 2.3.1
of Preferences 10.5
of ScriptEngine 8.2.6
of ScriptEngineManager 8.2.6
of Supplier 1.13
getActionCommand method
of ActionEvent 10.4.7
of ButtonModel 11.4.2, 11.8.4
of EventObject 10.4.7
getActionMap method
of JComponent 10.5
getAddress method 4.1.4
of InetAddress 4.6
getAdjustable, getAdjustmentType methods (AdjustmentEvent) 10.4.7
getAdvance method
of TextLayout 12.7.5
getAllByName method 4.1.4
of InetAddress 4.6
getAllowsChildren method
of TreeNode 12.7.5
getAncestorOfClass method 11.8.3
of SwingUtilities 11.8.4
getAscent method
of LineMetrics 10.5
of TextLayout 12.7.5
getAs*Xxx* method
of OptionalDouble 1.13
of OptionalInt 1.13
of OptionalLong 1.13
getAs*Xxx* methods
of Optional*Xxx* 1.12
getAtIndex method
of MemorySegment 13.8.3
getAttribute method 3.3
of Element 3.9
getAttributeCount method
of XMLStreamReader 3.9
getAttributeLocalName method
of XMLStreamReader 3.9
getAttributeName method
of XMLStreamReader 3.9
getAttributes method
of DocPrintJob 12.7.5

of Node 3.3, 3.9
of PrintService 12.7.5
getAttributeValue method
of XMLStreamReader 3.9
getAttributeXxx methods
of XMLStreamReader 3.7.2
getAuthority method 4.3.1
getAutoCommit method 5.9.3
of Connection 5.10
getAutoCreateRowSorter method
of JTable 12.7.5
getAvailableCurrencies method
of Currency 7.9
getAvailableFontFamilyNames method
of GraphicsEnvironment 10.3.3
getAvailableLocales method
of Collator 7.9
of NumberFormat 7.1.2, 7.2.1, 7.9
getAvailableZoneIds method
of ZoneId 6.5
getAverage method 1.9
of DoubleSummaryStatistics 1.13
of IntSummaryStatistics 1.13
of LongSummaryStatistics 1.13
getBackground method
of Component 10.5
getBinaryStream method 5.5.2
of Blob 5.10
getBlob method
of ResultSet 5.5.2, 5.10
getBlockSize method
of Cipher 9.4.4
getBoolean method
of Preferences 10.5
getBundle method 7.8.1
of ResourceBundle 7.9
getByName method 4.1.4
of InetAddress 4.6
getByteArray method 10.5
of Preferences 10.5
getBytes method 5.5.2
of Blob 5.10
getCandidateLocales method
of ResourceBundle.Control 7.8.1
getCategory method 12.7.5
of Attribute 12.7.5
getCellEditorValue method 12.3.3, 12.3.4
of CellEditor 12.7.5
getCellRenderer method 12.3.1
getCellSelectionEnabled method

of JTable 12.7.5
getCenterX method
of RectangularShape 10.5
getCenterX/Y methods
of RectangularShape 10.3.1
getCenterY method
of RectangularShape 10.5
getChannel method
of FileInputStream 2.7.7
of FileOutputStream 2.7.7
of RandomAccessFile 2.7.7
getChar method 2.4.1
getCharacterStream method 5.5.2
of Clob 5.10
getCharContent method
of SimpleJavaFileObject 8.2.6
getChild method 12.4.6
of TreeModel 12.7.5
getChildAt method
of TreeNode 12.7.5
getChildCount method
of TreeModel 12.4.6, 12.7.5
of TreeNode 12.7.5
getChildNodes method 3.3
of Node 3.9
getClassLoader method 9.1.1
of Class 9.4.4
getClickCount method 10.4.6, 10.4.7
of MouseEvent 10.5
getClip method 12.7.1
of Graphics 12.7.5
getClob method
of ResultSet 5.10
getClob method (ResultSet) 5.5.2
getCollationKey method 7.4
of Collator 7.9
getColorModel method 12.6.4
of BufferedImage 12.7.5
getColumn method 12.2.2
of TableColumnModel 12.7.5
getColumnClass method 12.2.1
of TableModel 12.7.5
getColumnCount method
of ResultSetMetaData 5.8, 5.10
of TableModel 12.1.2, 12.7.5
getColumnDisplaySize method
of ResultSetMetaData 5.10
getColumnLabel method
of ResultSetMetaData 5.10
getColumnModel method 12.2.2

of JTable 12.7.5
getColumnName method 12.1.2
of ResultSetMetaData 5.10
of TableModel 12.7.5
getColumnNumber method
of Diagnostic 8.2.6
of SAXParseException 3.9
getColumns method
of JTextField 11.8.4
getColumnSelectionAllowed method
of JTable 12.7.5
getColumnType method 12.3.3
getColumn*Xxx* methods
of ResultSetMetaData 5.8
getCommand method
of RowSet 5.10
getCompactNumberInstance method
of NumberFormat 7.9
getComponentPopupMenu method
of JComponent 11.8.4
getConcurrency method 5.6.1, 5.6.2
of ResultSet 5.10
getConnection method 5.3.4, 5.4.4, 5.10
of DriverManager 5.10
getConnectTimeout method
of URLConnection 4.6
getContent method
of URLConnection 4.6
getContentEncoding method
of URLConnection 4.6
getContentEncoding, getContentType methods
of URLConnection 4.3.2, 4.3.3
getContentEncoding, getContentType methods (URLConnection) 4.3.2
getContentLength method 4.3.2
of URLConnection 4.6
getContentType method
of URLConnection 4.6
getContext method
of ScriptEngine 8.2.6
getContextClassLoader method 9.1.2
of Thread 9.4.4
getCount method
of DoubleSummaryStatistics 1.13
of IntSummaryStatistics 1.13
of LongSummaryStatistics 1.13
getCountry method 1.9.3
of Locale 7.9
getCrc method
of ZipEntry 2.7.7
getCurrencyCode method

of Currency 7.9
getCurrencyInstance method 7.2.1, 7.2.3
of NumberFormat 7.9
getData method 3.3
of CharacterData 3.9
getDataElements method
of ColorModel 12.7.5
of Raster 12.6.4, 12.7.5
getDataSize method
of DataTruncation 5.10
getDate method
of ResultSet 5.4.1
of URLConnection 4.3.2, 4.6
getDayOfMonth method
of LocalDate 6.7
of ZonedDateTime 6.7
getDayOfWeek method
of LocalDate 6.7
of ZonedDateTime 6.7
getDayOfXxx methods
of LocalDate 6.2
getDayOfYear method
of LocalDate 6.7
of ZonedDateTime 6.7
getDecompositon method
of Collator 7.9
getDefault method 7.1.3
of FileSystems 2.7.7
of Locale 7.9
of SSLServerSocketFactory 4.6
of SSLSocketFactory 4.6
getDefaultEditor method
of JTable 12.7.5
getDefaultFractionDigits method
of Currency 7.9
getDefaultName method
of NameCallback 9.4.4
getDefaultRenderer method 12.3.2
of JTable 12.7.5
getDefaultToolkit method 10.2.2
of Toolkit 10.5
getDescent method
of LineMetrics 10.5
of TextLayout 12.7.5
getDescription method
of FileFilter 11.8.4
of FileView 11.8.4
getDiagnostics method
of DiagnosticCollector 8.2.6
getDisplayCountry method

of Locale 7.9
getDisplayCountry, getDisplayLanguage methods
of Locale 1.9.2
getDisplayLanguage method
of Locale 7.9
getDisplayName method
of DayOfWeek, Month 6.6, 7.3
of Locale 7.1.4, 7.2.1, 7.9
getDocumentElement method 3.3
of Document 3.9
getDoInput method
of URLConnection 4.6
getDoOutput method
of URLConnection 4.6
getDouble method
of ByteBuffer 2.4.1
of Preferences 10.5
of ResultSet 5.4.1
getEngineByExtension method
of ScriptEngineManager 8.2.6
getEngineByMimeType method
of ScriptEngineManager 8.2.6
getEngineByName method
of ScriptEngineManager 8.2.6
getEngineFactories method
of ScriptEngineManager 8.2.6
getEngineXxx methods
of ScriptEngineManager 8.2.1
getErrorCode method 5.4.3
of SQLException 5.10
getErrorStream method 4.3.3
of HttpURLConnection 4.6
getErrorWriter method
of ScriptContext 8.2.6
getExpiration method 4.3.2
of URLConnection 4.6
getExtensions method
of ScriptEngineFactory 8.2.6
getFamily method
of Font 10.5
getFields method 12.4.6
getFileName method 2.3.1
of Path 2.7.7
getFilePointer method 2.2.2
of RandomAccessFile 2.7.7
getFileSuffixes method
of ImageReaderWriterSpi 12.7.5
getFillsViewportHeight method
of JTable 12.7.5
getFirstChild method 3.3

of Node 3.9
getFloat method 2.4.1
of Preferences 10.5
getFont method
of Component 11.8.4
getFontMetrics method 10.3.3
of JComponent 10.5
getFontName method
of Font 10.5
getFontRenderContext method 12.5.8
of FontMetrics 10.5
of Graphics2D 10.3.3, 10.5, 12.7.5
getForeground method
of Component 10.5
getFormatNames method
of ImageReaderWriterSpi 12.7.5
getFragment method 4.3.1
getHeaderField method
of URLConnection 4.6
getHeaderFieldKey method
of URLConnection 4.6
getHeaderFields method
of URLConnection 4.6
getHeader*Xxx* methods
of URLConnection 4.3.2
getHeight method
of ImageReader 12.6.3, 12.7.5
of LineMetrics 10.5
of PageFormat 12.7.1, 12.7.5
of RectangularShape 10.3.1, 10.5
getHost method 4.3.1
getHostAddress method
of InetAddress 4.6
getHostName method
of InetAddress 4.6
getHour method
of LocalTime 6.7
of ZonedDateTime 6.7
getIcon method
of FileView 11.8.4
of JLabel 11.8.4
getIconImage method
of Frame 10.5
getIdentifier method
of RowFilter.Entry 12.7.5
getIfModifiedSince method
of URLConnection 4.6
getImage method
of ImageIcon 10.3.4, 10.5
getImageableHeight method

of PageFormat 12.7.5
getImageableWidth method
of PageFormat 12.7.5
getImageableX method
of PageFormat 12.7.5
getImageableXxx methods
of PageFormat 12.7.1
getImageableY method
of PageFormat 12.7.5
getImageReadersByFormatName method
of ImageIO 12.7.5
getImageReadersByMIMEType method
of ImageIO 12.7.5
getImageReadersBySuffix method
of ImageIO 12.7.5
getImageWritersByFormatName method
of ImageIO 12.7.5
getImageWritersByMIMEType method
of ImageIO 12.7.5
getImageWritersBySuffix method
of ImageIO 12.7.5
getImageXxxByXxx methods
of ImageIO 12.6.2
getIndex method
of DataTruncation 5.10
getIndexOfChild method 12.4.6
of TreeModel 12.7.5
getInheritsPopupMenu method
of JComponent 11.8.4
getInputMap method 10.4.5
of JComponent 10.5
getInputStream method
of Socket 4.1.2, 4.2.1, 4.6
of URLConnection 4.3.2, 4.3.3, 4.6
getInstance method
of AlphaComposite 12.5.9, 12.7.5
of Cipher 9.4.1, 9.4.4
of Collator 7.9
of Currency 7.2.3, 7.9
of KeyGenerator 9.4.4
of Locale 7.4
of MessageDigest 9.3.1, 9.4.4
getInt method
of ByteBuffer 2.4.1
of Preferences 10.5
of ResultSet 5.4.1
getInterface method
of Invocable 8.2.6
getISOCountries method 7.1.2
of Locale 7.9

getISOLanguages method 7.1.2
getItem, getItemSelectable methods
of ItemEvent 10.4.7
getItemAt method 11.4.4
getJavaFileForOutput method
of ForwardingJavaFileManager 8.2.6
getJavaFileObjectsFromFiles method
of StandardJavaFileManager 8.2.6
getJavaFileObjectsFromPaths method
of StandardJavaFileManager 8.2.6
getJavaFileObjectsFromStrings method
of StandardJavaFileManager 8.2.6
getJDBCMajorVersion method
of DatabaseMetaData 5.10
getJDBCMinorVersion method
of DatabaseMetaData 5.10
getKeys method 7.8.3
of ResourceBundle 7.9
getKeyStroke method 10.4.5
of KeyStroke 10.5
getKeyXxx methods
of KeyEvent 10.4.7
getKind method
of Diagnostic 8.2.6
getLanguage method
of Locale 7.9
getLargeUpdateCount method
of Statement 5.10
getLastChild method 3.3
of Node 3.9
getLastModifed method
of URLConnection 4.6
getLastModified method 4.3.2
getLastPathComponent method 12.4.2
of TreePath 12.7.5
getLastSelectedPathComponent method 12.4.2
of JTree 12.7.5
getLeading method
of LineMetrics 10.5
of TextLayout 12.7.5
getLength method
of Attributes 3.9
of NamedNodeMap 3.9
of NodeList 3.3, 3.9
getLineMetrics method 10.3.3
of Font 10.5
getLineNumber method
of Diagnostic 8.2.6
of SAXParseException 3.9
getLocale method

of MessageFormat 7.9
getLocalHost method 4.1.4
of InetAddress 4.6
getLocalName method
of Attributes 3.9
of Node 3.6, 3.9
of XMLStreamReader 3.9
getLong method 2.4.1
of Preferences 10.5
getMax method 1.9
of DoubleSummaryStatistics 1.13
of IntSummaryStatistics 1.13
of LongSummaryStatistics 1.13
getMaxConnections method
of DatabaseMetaData 5.10
getMaximumFractionDigits method
of NumberFormat 7.9
getMaximumIntegerDigits method
of NumberFormat 7.9
getMaxStatements method 5.4.2
of DatabaseMetaData 5.10
getMaxX method
of RectangularShape 10.5
getMaxY method
of RectangularShape 10.5
getMessage method
of Diagnostic 8.2.6
getMetaData method
of Connection 5.8, 5.10
of PreparedStatement 5.10
of ResultSet 5.8, 5.10
getMethodCallSyntax method 8.2.4
getMimeTypes method
of ImageReaderWriterSpi 12.7.5
of ScriptEngineFactory 8.2.6
getMin method
of DoubleSummaryStatistics 1.13
of IntSummaryStatistics 1.13
of LongSummaryStatistics 1.13
getMinimumFractionDigits method
of NumberFormat 7.9
getMinimumIntegerDigits method
of NumberFormat 7.9
getMinute method
of LocalTime 6.7
of ZonedDateTime 6.7
getMinX method
of RectangularShape 10.5
getMinY method
of RectangularShape 10.5

getModel method
of RowFilter.Entry 12.7.5
getModifiers method
of ActionEvent 10.4.7
getMonth method
of LocalDate 6.7
of ZonedDateTime 6.7
getMonthValue method
of LocalDate 6.7
of ZonedDateTime 6.7
getMoreResults method 5.5.4
of Statement 5.10
getName method
of Attribute 12.7.5
of Field 12.4.6
of FileView 11.8.4
of Font 10.5
of NameCallback 9.4.4
of Principal 9.4.4
of PrintService 12.7.3
of XMLStreamReader 3.9
of ZipEntry 2.7.7
getNames method
of ScriptEngineFactory 8.2.6
getNamespaceURI method 3.6
of Node 3.9
getNano method
of LocalTime 6.7
of ZonedDateTime 6.7
getNewState, getOldState methods
of WindowEvent 10.4.7
getNextEntry method 2.2.3
of ZipInputStream 2.7.7
getNextException method 5.4.3
of SQLException 5.10
getNextSibling method 3.3
of Node 3.9
getNextWarning method
of SQLWarning 5.10
getNodeName method
of Node 3.9
getNodeValue method
of Node 3.9
getNodeXxx methods (Node) 3.3, 3.6
getNumberInstance method 7.2.1
of NumberFormat 7.9
getNumericCode method
of Currency 7.9
getNumericCodeAsString method
of Currency 7.9

getNumImages method
of ImageReader 12.7.5
getNumThumbnails method
of ImageReader 12.7.5
getNumXxx methods
of ImageReader 12.6.3
getObject method
of ResourceBundle 7.8.3, 7.9
of ResultSet 5.4.1, 5.10
getOffset method
of ZonedDateTime 6.7
getOppositeWindow method 10.4.7
getOrientation method
of PageFormat 12.7.5
getOriginatingProvider method
of ImageReader 12.6.2, 12.7.5
of ImageWriter 12.7.5
getOutputSize method
of Cipher 9.4.4
getOutputStream method
of Socket 4.2.1, 4.6
of URLConnection 4.3.2, 4.3.3, 4.6
getOwner method 2.3.5
getPageCount method 12.7.2
getPageSize method
of CachedRowSet 5.10
getPaint method
of Graphics2D 10.5
getParameter method
of DataTruncation 5.10
getParameterClassName method
of ParameterMetaData 5.10
getParameterCount method
of ParameterMetaData 5.10
getParameterMode method
of ParameterMetaData 5.10
getParameterType method
of ParameterMetaData 5.10
getParameterTypeName method
of ParameterMetaData 5.10
getParent method
of ClassLoader 9.4.4
of Path 2.3.1, 2.7.7
of TreeNode 12.4.3, 12.7.5
getParentNode method
of Node 3.9
getPassword method
of JPasswordField 11.8.4
of PasswordCallback 9.4.4
of RowSet 5.10

getPath method
of FileSystem 2.3.9, 2.7.7
of TreeSelectionEvent 12.7.5
of URI 4.3.1
getPathMatcher method
of FileSystem 2.7.7
getPaths method 12.4.5
of TreeSelectionEvent 12.7.5
getPathToRoot method 12.4.2
getPercentInstance method 7.2.1
of NumberFormat 7.9
getPixel method
of Raster 12.7.5
getPixel, getPixels methods
of Raster 12.6.4
getPixels method
of Raster 12.7.5
getPoint method 10.4.7
of MouseEvent 10.5
getPointCount method 12.5.3
getPort method 4.3.1
getPrecision method
of ParameterMetaData 5.10
getPredefinedCursor method 10.4.6
getPreferredSize method 10.3
of Component 10.5
getPreviousSibling method
of Node 3.9
getPrincipals method
of Subject 9.4.4
getPrinterJob method 12.7.1
of PrinterJob 12.7.5
getPrintService method 12.7.4
of StreamPrintServiceFactory 12.7.5
getPrompt method
of NameCallback 9.4.4
of PasswordCallback 9.4.4
getQName method
of Attributes 3.9
getQuery method 4.3.1
getRaster method 12.6.4
of BufferedImage 12.7.5
getReader method
of ScriptContext 8.2.6
getReaderFileSuffixes method
of ImageIO 12.7.5
getReaderFormatNames method
of ImageIO 12.7.5
getReaderMIMETypes method
of ImageIO 12.7.5

getReaderXxx methods
of ImageIO 12.6.2
getReadTimeout method
of URLConnection 4.6
getRequestProperties method
of URLConnection 4.6
getResponseCode method 4.3.3
getResultSet method
of Statement 5.10
getRGB method
of Color 12.7.5
of ColorModel 12.6.4, 12.7.5
getRoot method
of Path 2.3.1, 2.7.7
of TreeModel 12.4.6, 12.7.5
getRootPane method 11.8.3
of JComponent 11.8.4
getRotateInstance method 12.5.7
of AffineTransform 12.7.5
getRow method 5.6.1
of ResultSet 5.10
getRowCount method 12.1.2
of TableModel 12.7.5
getRowHeight method
of JTable 12.7.5
getRowMargin method
of JTable 12.7.5
getRowSelectionAllowed method
of JTable 12.7.5
getSavepointId method
of Savepoint 5.10
getSavepointName method
of Savepoint 5.10
getScale method
of ParameterMetaData 5.10
getScaleInstance method 12.5.7
of AffineTransform 12.7.5
getScreenSize method 10.2.2
of Toolkit 10.5
getScrollAmount method 10.4.7
getSecond method
of LocalTime 6.7
of ZonedDateTime 6.7
getSeconds method 6.1
getSelectedColumns method 12.2.5, 12.2.8
getSelectedFile method
of JFileChooser 11.8.4
getSelectedFile/Files methods
of JFileChooser 11.8.4
getSelectedFiles method

of JFileChooser 11.8.4
getSelectedItem method 11.4.4
of JComboBox 11.8.4
getSelectedObjects method 11.4.2
getSelectedRows method 12.2.5
getSelection method 11.4.2
of ButtonGroup 11.8.4
getSelectionModel method
of JTable 12.7.5
getSelectionPath method 12.4.2
of JTree 12.7.5
getSelectionPaths method 12.4.5
of JTree 12.7.5
getShearInstance method 12.5.7
of AffineTransform 12.7.5
getShort method 2.4.1
getSize method
of Component 10.5
of ZipEntry 2.7.7
getSource method 10.4.7
of Diagnostic 8.2.6
getSQLState method 5.4.3
of SQLException 5.10
getSQLStateType method 5.4.3
getStandardFileManager method
of JavaCompiler 8.2.6
getStateChange method 10.4.7
getStrength method
of Collator 7.9
getString method
of MemorySegment 13.8.3
of ResourceBundle 7.8.2, 7.9
of ResultSet 5.4.1
getStringArray method
of ResourceBundle 7.9
getStringBounds method 10.3.3
of Font 10.5
GetStringUTFChars 13.1
getStringValue method
of RowFilter.Entry 12.7.5
getSubject method
of LoginContext 9.4.4
getSubString method 5.5.2
of Clob 5.10
getSum method
of DoubleSummaryStatistics 1.13
of IntSummaryStatistics 1.13
of LongSummaryStatistics 1.13
getSymbol method
of Currency 7.9

getSystemClassLoader method
of ClassLoader 9.4.4
getSystemJavaCompiler method 8.1.1
getTableCellEditorComponent method 12.3.4
of TableCellEditor 12.7.5
getTableCellRendererComponent method 12.3.1
of TableCellRenderer 12.7.5
getTableHeader method 12.1.1
getTableName method
of CachedRowSet 5.10
getTables method 5.8
of DatabaseMetaData 5.10
getTagName method 3.3, 3.6
of Element 3.9
getTask method 8.1.2
of JavaCompiler 8.2.6
getText method
of JLabel 11.8.4
of JTextComponent 11.3.1, 11.8.4
of XMLStreamReader 3.9
getTimeZone method 6.7
getTitle method
of Frame 10.5
getTransferSize method
of DataTruncation 5.10
getTranslateInstance method 12.5.7, 12.5.8
of AffineTransform 12.7.5
getTreeCellRendererComponent method
of DefaultTreeCellRenderer 12.4.4
of TreeCellRenderer 12.4.4, 12.7.5
getType method
of Field 12.4.6
of ResultSet 5.6.1, 5.10
getTypeDescription method 11.8.4
of FileView 11.8.4
getUpdateCount method 5.5.4
of Statement 5.10
getURI method
of Attributes 3.9
getURL method
of RowSet 5.10
getURLs method 9.1.2
getUserInfo method 4.3.1
getUsername method
of RowSet 5.10
getValue method
of Action 10.4.5, 10.5
of AdjustmentEvent 10.4.7
of Attributes 3.9
of Copies 12.7.5

of RowFilter.Entry 12.7.5
getValueAt method 12.1.2, 12.3.3
of TableModel 12.7.5
getValueCount method
of RowFilter.Entry 12.7.5
getVendorName method
of IIOServiceProvider 12.7.5
getVendorName, getVersion methods
of IIOServiceProvider 12.6.2
getVersion method
of IIOServiceProvider 12.7.5
getWarnings method
of Connection 5.10
getWheelRotation method 10.4.7
getWidth method
of ImageReader 12.6.3, 12.7.5
of PageFormat 12.7.1, 12.7.5
of Rectangle2D 10.3.1
of RectangularShape 10.3.1, 10.5
getWindow method 10.4.7
getWriter method
of ScriptContext 8.2.6
getWriterFileSuffixes method
of ImageIO 12.7.5
getWriterFormatNames method
of ImageIO 12.7.5
getWriterMIMETypes method
of ImageIO 12.7.5
getWriterXxx methods
of ImageIO 12.6.2
getX method
of MouseEvent 10.5
of RectangularShape 10.5
getX/Y methods
of MouseEvent 10.4.6, 10.4.7
get*Xxx* method
of ByteBuffer 2.7.7
of Period 6.7
of ResultSet 5.10
getY method
of MouseEvent 10.5
of RectangularShape 10.5
getYear method
of LocalDate 6.7
of ZonedDateTime 6.7
GIF 12.6.1
animated 12.6.3
image manipulations on 12.6.5
printing 12.7.3
GlassFish server 5.10

Glob patterns 2.3.7, 2.3.8
Global arena 13.3
GlobalSign 9.3.5
GMail 4.6
Gnu C compiler 13.1
Google Maps 4.3.3
GradientPaint class 12.5.6, 12.7.5
constructor 12.5.6, 12.7.5
cyclic parameter 12.5.6
Grapheme clusters 1.3
Graphic Java™ (Geary) 12.1, 12.4
Graphical User Interface 10
components of 11
choice components 11.4
dialog boxes 11.8
menus 11.5
text input 11.3
toolbars 11.5.7
tooltips 11.5.8
events in 10.4
keyboard focus in 10.4.5
layout of 11.2, 11.6
Graphics class 10.3.1, 10.3.4, 10.5, 12.5, 12.5.1, 12.7.5
copyArea method 10.5
drawImage method 10.3.4, 10.5
drawXxx, fillXxx methods 12.5.2
get/setClip methods 12.5.8, 12.7.1
getClip method 12.7.5
setClip method 12.7.5
Graphics editor applications 10.4.6
Graphics2D class 10.3.1, 10.5, 12.5.1, 12.7.5
clip method 12.5.1, 12.5.8, 12.7.1, 12.7.5
draw method 10.3.1, 12.5.1, 12.5.2, 12.5.4, 12.5.5, 12.7.5
drawString method 10.5
fill method 10.3.2, 10.5, 12.5.1, 12.5.4, 12.7.5
getFontRenderContext method 10.3.3, 10.5, 12.5.8, 12.7.5
getPaint method 10.5
rotate method 12.5.7, 12.7.5
scale method 12.5.7, 12.7.5
setComposite method 12.5.1, 12.5.9, 12.7.5
setPaint method 10.3.2, 10.5, 12.5.1, 12.5.6, 12.7.5
setRenderingHint, setRenderingHints methods 12.5.1
setStroke method 12.5.1, 12.5.5, 12.7.5
setTransform method 12.5.7, 12.7.5
shear method 12.5.7, 12.7.5
transform method 12.5.1, 12.5.7, 12.7.5
translate method 12.5.7, 12.7.2, 12.7.5
GraphicsEnvironment class 10.3.3
Greenwich Royal Observatory 6.1, 6.5
GregorianCalendar class

toZonedDateTime method 6.7
Grid bag layout 11.6
Grid layout 11.2.3
GridBagConstraints class 11.6.1, 11.8.4
anchor field 11.8.4
anchor, fill parameters 11.6.4
Constructor 11.8.4
fill field 11.8.4
gridwidth field 11.8.4
gridx field 11.8.4
gridx/y, gridwidth/height parameters 11.6.2
helper class for 11.6.8
insets field 11.8.4
insets parameter 11.6.5
ipadx field 11.8.4
weightx field 11.8.4
weightx/y parameters 11.6.3
GridLayout class 11.2.1, 11.2.3, 11.8.4
Constructor 11.8.4
gridwidth field
of GridBagConstraints 11.8.4
gridx field
of GridBagConstraints 11.8.4
Groovy 8.2, 8.2.1
Group layout 11.6
group method
of MatchResult 2.7.7
groupElement method
of MemoryLayout.PathElement 13.8.3
Grouping 1.9.3
classifier functions of 1.9.3
reducing to numbers 1.9.4
groupingBy method 1.9.3
of Collectors 1.13
groupingByConcurrent method 1.9.3, 1.13
of Collectors 1.13
GUI. See Graphical User Interface

H

Half-closing connections 4.2.3
Handbook of Applied Cryptography, The (Menezes et al.) 9.3.3
handle method
of CallbackHandler 9.4.4
handleGetObject method 7.8.3
of ResourceBundle 7.9
Handles 12.4.1, 12.4.4
HashSet class
readObject, writeObject methods 2.6.4
HashXxxAttributeSet classes 12.7.1, 12.7.5

Haskell 8.2
hasMatch method
of Matcher 2.7.7
hasNext method
of XMLStreamReader 3.9
hasRemaining method
of Buffer 2.7.7
Header information, from server 4.3.2
header method
of HttpRequest.Builder 4.6
Headers 4.4.3
Headers (Swing tables)
rendering 12.3.2
scrolling 12.1.1
headers method
of HttpResponse 4.6
Height 10.3.3
Helper classes 11.6.8
Hex editors
creating class files in 9.1.5
modifying bytecodes with 9.1.5
Hidden commands, in XML comments 3.2
Hosts 4.1.4
HTML
attributes in 3.1, 3.2
end and empty tags in 3.1
forms in 4.3.3
generating from XML files 3.9
in labels 11.3.2
mixing with JSP 8.1.6
printing 12.7.3
tables in 11.6.1
vs. XML 3.1
HTML editors 11.1
HTTP 5.1.2
redirects between HTTPS and 4.3.3
request headers in 4.3.2
http: scheme 4.3.1
HttpClient class 4.4, 4.6
enabling logging for 4.4.4
newBuilder method 4.4.1, 4.4.4, 4.6
newHttpClient method 4.4.1, 4.6
send method 4.6
sendAsync method 4.4.4, 4.6
HttpClient.Builder interface 4.6
build method 4.4.1, 4.4.2, 4.6
executor method 4.6
followRedirects method 4.4.1, 4.6
HttpHeaders class 4.6
firstValue method 4.4.3, 4.6

map method 4.4.3, 4.6
HttpRequest class 4.6
newBuilder method 4.4.2, 4.4.4, 4.6
HttpRequest.Builder interface 4.6
build method 4.6
DELETE method 4.6
GET method 4.4.2, 4.6
header method 4.6
POST method 4.4.2, 4.6
PUT method 4.6
uri method 4.4.2, 4.4.4, 4.6
HttpResponse class
methods of 4.4.3
HttpResponse interface 4.6
body method 4.6
headers method 4.6
statusCode method 4.6
HttpResponse.BodyHandlers class
discarding method 4.4.3
ofString method 4.4.3, 4.4.4
HTTPS 4.3.3
https: scheme 4.3.1
HttpURLConnection class 4.3.3, 4.6
getErrorStream method 4.3.3, 4.6
getResponseCode method 4.3.3
setInstanceFollowRedirects method 4.3.3

I

I (int), type code 2.6.2
IANA 6.5
IBM 3
DB2 database 5.3
DOM parser 3.3
IBM437 encoding 7.7.3
ICC profiles 12.6.4
Icons
in column headers 12.3.2
in menu items 11.5.2
in sliders 11.4.5
in table cells 12.3
in trees 12.4.1, 12.4.4
ID, IDREF, IDREFS attribute types (DTDs) 3.4.1
identity method
of Function 1.9.2
Identity values 1.10
IDs, uniqueness of 3.4.1, 3.4.3
IETF BCP 47 7.1.2
IFC 10.1
ifPresent method

of Optional 1.7.2, 1.13
of OptionalDouble 1.13
of OptionalInt 1.13
of OptionalLong 1.13
ifPresentOrElse method 1.7.2
of Optional 1.13
IIOImage class 12.6.3, 12.7.5
Constructor 12.7.5
IIOServiceProvider class 12.7.5
getVendorName method 12.7.5
getVersion method 12.7.5
getXxx methods 12.6.2
IllegalAccessException class 12.4.6
IllegalStateException class 1.9.2, 12.6.3
ImageIcon class 10.2.2, 10.5
Constructor 10.5
getImage method 10.3.4, 10.5
ImageInputStream interface 12.6.3
“Seek forward only” mode 12.6.3
ImageIO class 12.7.5
createImageInputStream method 12.7.5
createImageOutputStream method 12.7.5
createImageXxxStream methods 12.6.3
determining image type 12.6.1, 12.6.2
getImageReadersByFormatName method 12.7.5
getImageReadersByMIMEType method 12.7.5
getImageReadersBySuffix method 12.7.5
getImageWritersByFormatName method 12.7.5
getImageWritersByMIMEType method 12.7.5
getImageWritersBySuffix method 12.7.5
getImageXxxByXxx methods 12.6.2
getReaderFileSuffixes method 12.7.5
getReaderFormatNames method 12.7.5
getReaderMIMETypes method 12.7.5
getReaderXxx, getWriterXxx methods 12.6.2
getWriterFileSuffixes method 12.7.5
getWriterFormatNames method 12.7.5
getWriterMIMETypes method 12.7.5
read method 12.7.5
read, write methods 12.6.1
write method 12.7.5
ImageOutputStream interface 12.6.3
ImageReader class 12.6.2, 12.7.5
getHeight method 12.6.3, 12.7.5
getNumImages method 12.7.5
getNumThumbnails method 12.7.5
getNumXxx methods 12.6.3
getOriginatingProvider method 12.6.2, 12.7.5
getWidth method 12.6.3, 12.7.5
read method 12.7.5

readThumbnail method 12.7.5
setInput method 12.7.5
ImageReaderWriterSpi class 12.7.5
getFileSuffixes method 12.7.5
getFormatNames method 12.7.5
getMIMETypes method 12.7.5
Images
blurring 12.6.5
color values of 12.6.4
edge detection of 12.6.5
filtering 12.6.5
getting size of, before reading 12.6.3
incremental rendering of 12.6.4
manipulating 12.6.4
metadata in 12.6.3
multiple, in a file 12.6.3
printing 12.7.1, 12.7.3, 12.7.4
raster 12.6
constructing from pixels 12.6.4
readers/writers for 12.6.1
rotating 12.6.5
superimposing 12.5.9
thumbnails for 12.6.3
vector 12.5
Images, displaying 10.3.4
ImageWriter class 12.6.2, 12.6.3, 12.7.5
canInsertImage method 12.6.3, 12.7.5
getOriginatingProvider method 12.7.5
setOutput method 12.7.5
write method 12.7.5
write, writeInsert methods 12.6.3
writeInsert method 12.7.5
#IMPLIED 3.4.1
import keyword 9.1.3
importPreferences method 10.5
of Preferences 10.5
include method
of RowFilter 12.2.7, 12.7.5
Incremental rendering 12.6.4
Indexed color model 12.6.5
IndexOutOfBoundsException class 12.6.3
InetAddress class 4.6
getAddress method 4.6
getAllByName method 4.6
getByName method 4.6
getHostAddress method 4.6
getHostName method 4.6
getLocalHost method 4.6
get*Xxx* methods 4.1.4
InetSocketAddress class 4.6

Constructor 4.6
isUnresolved method 4.6
Inheritance trees 5.4.3
init method
of Cipher 9.4.4
of KeyGenerator 9.4.4
initialize method
of LoginModule 9.4.4
initializer method
of Gatherer 1.13
Input dialogs 11.8.1
Input maps 10.4.5
Input streams 2.1
as input source 3.3
buffered 2.1.3
byte processing in 2.1.3
byte-oriented 2.1
chaining 2.1.3
closing 2.1.1
filters for 2.1.3
hierarchy of 2.1.2
keeping open 4.2.3
objects in 2.6
redirecting 8.2.3
Unicode 2.1
InputSource class 3.9
Constructor 3.9
InputStream class 2.1.1, 2.1.2, 2.7.7
available method 2.1.1, 2.7.7
close method 2.1.1, 2.7.7
mark method 2.7.7
markSupported method 2.7.7
nullInputStream method 2.7.7
read method 2.1.1, 2.7.7
readAllBytes method 2.7.7
readAllBytes, readNBytes methods 2.1.1
readNBytes method 2.7.7
reset method 2.7.7
skip method 2.7.7
skipNBytes method 2.7.7
transferTo method 2.7.7
InputStreamReader class 2.1.4
INSERT 5.2
executing 5.4.1
in batch updates 5.9.3
vs. methods of ResultSet 5.6.2
insert method
of JMenu 11.8.4
insertItemAt method 11.4.4
of JComboBox 11.8.4

insertNodeInto method 12.4.2
of DefaultTreeModel 12.7.5
insertRow method 5.6.2
of ResultSet 5.10
insertSeparator method
of JMenu 11.8.4
insets field
of GridBagConstraints 11.8.4
INSTANCE 2.6.6
Instant class 6.1, 6.7
compareTo method 6.1
equals method 6.1
from method 6.7
immutability of 6.1
legacy classes and 6.7
minus method 6.7
minus*Xxx* method 6.7
now method 6.1, 6.7
plus method 6.7
plus*Xxx* method 6.7
int type
printing 2.1.6
storing 2.2.1
streams of 1.12
type code for 2.6.2
writing in binary format 2.2.1
IntBuffer class 2.4.2
INTEGER 5.2, 5.9.4
IntegerSyntax class 12.7.5
integrate method
of Gatherer.Integrator 1.13
Intel processor, little-endian order in 2.2.1
IntelliJ IDE 2.6.4
Interfaces
accessing script classes with 8.2.4
implementing in script engines 8.2.4
methods in
do-nothing 10.4.4
Intermittent bugs 10.2.1
Internal padding 11.6.5
Internationalization 7, 7.9
Internet Engineering Task Force 7.1.2
Interpolation 12.6.5
for gradients 12.5.6
strategies of 12.6.5
when transforming images 12.6.5
Interruptible sockets 4.2.4
intersect method
of Area 12.5.4, 12.7.5
ints method

of RandomGenerator 1.12, 1.13
IntStream interface 1.12, 1.13
average method 1.12, 1.13
boxed method 1.12, 1.13
mapToInt method 1.12
max method 1.13
max, min methods 1.12
min method 1.13
of method 1.12, 1.13
range method 1.13
range, rangeClosed methods 1.12
rangeClosed method 1.13
sum method 1.13
sum, summaryStatistics methods 1.12
summaryStatistics method 1.13
toArray method 1.12, 1.13
IntSummaryStatistics class 1.9, 1.12, 1.13
getAverage method 1.13
getCount method 1.13
getMax method 1.13
getMin method 1.13
getSum method 1.13
intValue method 7.2.1
Invalid pointers (C, C++) 13
InvalidObjectException class 2.6.7
InvalidPathException class 2.3.1
Invocable interface 8.2.4, 8.2.6
getInterface method 8.2.6
invokeFunction method 8.2.6
invokeMethod method 8.2.6
invokeXxx methods 8.2.4
invoke method
of MethodHandle 13.8.3
invokeFunction method
of Invocable 8.2.6
invokeMethod method
of Invocable 8.2.6
IOException class 2.1.2, 2.1.5, 4.1.2
IP addresses 4.1.1, 4.1.4
ipadx field
of GridBagConstraints 11.8.4
IPP (Internet Printing Protocol) 1.1 12.7.5
IPv6 addresses 4.1.4
isActionKey method 10.4.7
isAfter method
of LocalDate 6.7
of LocalTime 6.7
of ZonedDateTime 6.7
isAfterLast method 5.6.1
of ResultSet 5.10

isBefore method
of LocalDate 6.7
of LocalTime 6.7
of ZonedDateTime 6.7
isBeforeFirst method 5.6.1
of ResultSet 5.10
isCellEditable method
of AbstractCellEditor 12.3.4
of AbstractTableModel 12.3.3
of CellEditor 12.7.5
of DefaultTableModel 12.3.3
of TableModel 12.3.3, 12.7.5
isCharacters method
of XMLStreamReader 3.9
isClosed method
of ResultSet 5.10
of Socket 4.6
of Statement 5.10
isConnected method
of Socket 4.6
isDefaultButton method
of JButton 11.8.4
isDirectory method
of BasicFileAttributes 2.7.7
of Files 2.7.7
of ZipEntry 2.7.7
isEchoOn method
of PasswordCallback 9.4.4
isEditable method
of JComboBox 11.8.4
of JTextComponent 11.8.4
isEmpty method
of Optional 1.13
isEnabled method 10.4.5
of Action 10.5
isEndElement method
of XMLStreamReader 3.9
isExecutable method 2.3.5
of Files 2.7.7
isFirst method 5.6.1
of ResultSet 5.10
isGroupingUsed method
of NumberFormat 7.9
isHidden method 2.3.5
of Files 2.7.7
isIgnoringElementContentWhitespace method
of DocumentBuilderFactory 3.9
isInputShutdown method
of Socket 4.6
isLast method 5.6.1

of ResultSet 5.10
isLeaf method
of DefaultTreeModel 12.4.1
of TreeModel 12.4.6, 12.7.5
of TreeNode 12.4.1, 12.7.5
isLeapYear method
of LocalDate 6.7
isNamespaceAware method
of DocumentBuilderFactory 3.9
of SAXParserFactory 3.9
isNegative method
of Duration 6.7
isNullable method
of ParameterMetaData 5.10
ISO 216 7.8.3
ISO 3166-1 7.1.2
ISO 4217 7.2.2, 7.2.3
ISO 639-1 7.1.2
ISO 8601 5.5.3
ISO 8859-1 2.1.4, 2.1.8
isOther method
of BasicFileAttributes 2.7.7
isOutputShutdown method
of Socket 4.6
isParseIntegerOnly method
of NumberFormat 7.9
isPopupTrigger method
of JPopupMenu 11.8.4
of MouseEvent 11.8.4
isPresent method 1.7.4, 1.7.7
of Optional 1.13
isReadable method 2.3.5
of Files 2.7.7
isRegularFile method
of BasicFileAttributes 2.7.7
of Files 2.3.5, 2.7.7
isRejecting method
of Gatherer.Downstream 1.13
isResizable method
of Frame 10.5
isSelected method
of AbstractButton 11.8.4
of JCheckBox 11.4.1, 11.8.4
isShared method 2.5
isStartElement method
of XMLStreamReader 3.9
isSymbolicLink method
of BasicFileAttributes 2.7.7
of Files 2.3.5, 2.7.7
isTemporary method 10.4.7

isTraversable method 11.8.4
of FileView 11.8.4
isUnresolved method
of InetSocketAddress 4.6
isValidating method
of DocumentBuilderFactory 3.9
of SAXParserFactory 3.9
isVisible method
of Component 10.5
isWhiteSpace method
of XMLStreamReader 3.9
isWritable method 2.3.5
of Files 2.7.7
isZero method
of Duration 6.7
item method
of NamedNodeMap 3.9
of NodeList 3.3, 3.4.1, 3.9
ItemEvent class 10.4.7
get*Xxx* methods 10.4.7
ItemListener interface, itemStateChanged method 10.4.7
ItemSelectable interface, getSelectedObjects method 11.4.2
Iterable interface 1.13, 2.3.8, 8.1.2
spliterator method 1.13
iterate method
of Stream 1.2, 1.5, 1.12, 1.13
Iterator interface 5.4.1
iterator method
of BaseStream 1.13
of SQLException 5.4.3, 5.10
of Stream 1.8
Iterators 1.8
converting to streams 1.2, 1.13

J

J (long), type code 2.6.2
JAAS 9.2.1
configuration files in 9.2.1
login modules in 9.2.2
JAR files
adding to class path 9.1.2
automatic registration in 5.3.4
for plugins 9.1.2
manifest of 2.2.3
resources in 7.8
signing 9.3.4, 9.3.7
jar: scheme 4.3.1
Jarsigner 9.3.4
Jar*Xxx*Stream classes 2.2.3

Java
-noverify option 9.1.5
internationalization support in 7
platform-independent 2.2.1
specifying locales in 7.1.3
versions of 10.1, 11.6
vs. SQL 5.5.1
Java 2D API 12.5
affine transformations in 12.5.7
colors in 12.6.4
constructive area geometry operations in 12.5.4
features supported in 12.5.1
filters in 12.6.5
paint in 12.5.6
printing in 12.7.2
rendering pipeline in 12.5.1
sample values in 12.6.4
shape classes in 12.5.2, 12.5.3
strokes in 12.5.5
transparency in 12.5.9
Java 2D library 10.3.1
floating-point coordinates in 10.3.1
Java Bug Database 12.6.5
Java Development Kit (JDK)
documentation in 10.4.5
fonts shipped with 10.3.3
Java EE 5.1.2
Java look-and-feel 10.4.5
Java Native Interface 13
Java Plug-in, loading signed code 9.3.7
Java Virtual Machine Specification 9.1.5
java.awt.geom package 2.6.4
java.base package 9.1.1
java.datatransfer package 9.1.1
java.desktop package 9.1.1
java.instrument package 9.1.1
java.io package 2.1.8
java.logging package 9.1.1
java.management package 9.1.1
java.management.rmi package 9.1.1
java.naming package 9.1.1
java.net package
socket connections in 4.1.2
supporting IPv6 addresses in 4.1.4
URLs vs. URIs in 4.3.1
java.net.http package 4.4
java.nio package 4.2.4
memory mapping in 2.4.1
java.prefs package 9.1.1
java.rmi package 9.1.1

java.security package 9.3
java.security.sasl package 9.1.1
java.sql package 6.7
java.time package 7.3
java.xml package 9.1.1
JavaBeans 11.8.4
Javac
byte order mark in 7.7.4
passing arguments to 8.1.1
JavaCompiler interface 8.2.6
getStandardFileManager method 8.2.6
getTask method 8.1.2, 8.2.6
JavaCompiler.CompilationTask interface 8.2.6
call method 8.2.6
JavaFileObject interface 8.1.2
JavaFX 10.1
JavaScript 8.2, 8.2.6
javax.imageio package 12.6.1
javax.sql package 5.10
javax.sql.rowset package 5.7.1
javax.swing package 10.2.1
javax.xml.catalog.files system property 3.4.1
JAXP (Java API for XML Processing) 3.3
JAXP (Java API for XML Processing) library 9.1.2
JButton class 10.4.2, 10.4.5, 10.5, 11.1, 11.8.4
Constructor 10.5
isDefaultButton method 11.8.4
Jchar 13.1
JCheckBox class 11.4.1, 11.8.4, 12.3.3
Constructor 11.8.4
is/setSelected methods 11.4.1
isSelected method 11.8.4
setSelected method 11.8.4
JCheckBoxMenuItem class 11.5.3, 11.8.4
Constructor 11.8.4
JComboBox class 10.4.7, 11.4.4, 11.8.4, 12.3.3
addItem method 11.4.4, 11.8.4
getItemAt method 11.4.4
getSelectedItem method 11.4.4, 11.8.4
insertItemAt method 11.4.4, 11.8.4
isEditable method 11.8.4
removeAllItems method 11.8.4
removeAllItems, removeItemAt methods 11.4.4
removeItem method 11.4.4, 11.8.4
removeItemAt method 11.8.4
setEditable method 11.4.4, 11.8.4
setModel method 11.4.4
JComponent class 10.3, 10.5, 11.8.4, 12.7.5
action maps 10.4.5
get/setComponentPopupMenu methods 11.5.4

get/setInheritsPopupMenu methods 11.5.4
getActionMap method 10.5
getComponentPopupMenu method 11.8.4
getFontMetrics method 10.3.3, 10.5
getInheritsPopupMenu method 11.8.4
getInputMap method 10.4.5, 10.5
getRootPane method 11.8.3, 11.8.4
input maps 10.4.5
paint method 12.3.1, 12.5.1
paintComponent method 10.3, 10.3.3, 10.3.4, 10.5, 12.5.1
putClientProperty method 12.4.1, 12.7.5
revalidate method 11.3.1, 11.8.4
setBorder method 11.4.3, 11.8.4
setComponentPopupMenu method 11.8.4
setFont method 11.8.4
setInheritsPopupMenu method 11.8.4
setToolTipText method 11.8.4
JDBC 5, 5.10
configuration of 5.3
debugging 5.3.4
design of 5.1
specification for 5.1.1
tracing 5.3.4
uses of 5.1.2
versions of 5
JDBC API Tutorial and Reference (Fisher et al.) 5.6.2, 5.9.4
JDBC drivers
escape syntax in 5.5.3
JAR files for 5.3.2
registering classes for 5.3.4
scrollable/updatable result sets in 5.6.1
types of 5.1.1
JDBC/ODBC bridge 5.1.1
JdbcRowSet interface 5.7.1
JDialog class 11.8.2, 11.8.4
JDialog method 11.8.4
setDefaultCloseOperation method 11.8.2
setVisible method 11.8.2, 11.8.3
JDialog method
of JDialog 11.8.4
JDK
DOM parser 3.3
keytool program 9.3.4
serialver program 2.6.7
SunJCE ciphers 9.4.1
jdk.incubator.http package 4.4
JEditorPane class 11.3.5
JEP 290, JEP 415 2.6.8
jextract tool 13.6
JFileChooser class 11.8.4

addChoosableFileFilter method 11.8.4
Constructor 11.8.4
getSelectedFile method 11.8.4
getSelectedFile/Files methods 11.8.4
getSelectedFiles method 11.8.4
resetChoosableFileFilters method 11.8.4
resetChoosableFilters method 11.8.4
setAcceptAllFileFilterUsed method 11.8.4
setAccessory method 11.8.4
setCurrentDirectory method 11.8.4
setFileFilter method 11.8.4
setFileSelectionMode method 11.8.4
setFileView method 11.8.4
setMultiSelectionEnabled method 11.8.4
setSelectedFile method 11.8.4
setSelectedFile/Files methods 11.8.4
setSelectedFiles method 11.8.4
showDialog method 11.8.3, 11.8.4
showOpenDialog method 11.8.4
showSaveDialog method 11.8.4
showXxxDialog methods 11.8.4
JFrame class 10.2, 10.2.2, 10.5, 11.2.1, 11.8.4
add method 10.5
internal structure of 10.3
setJMenuBar method 11.5.1, 11.8.4
JLabel class 11.3.2, 11.8.4, 12.4.4
Constructor 11.8.4
getIcon method 11.8.4
getText method 11.8.4
setIcon method 11.8.4
setText method 11.8.4
JMenu class 11.8.4
add method 11.5.1, 11.8.4
addSeparator method 11.5.1, 11.8.4
Constructor 11.8.4
insert method 11.8.4
insertSeparator method 11.8.4
remove method 11.8.4
JMenuBar class 11.5.1
JMenuItem class 11.8.4
Constructor 11.8.4
setAccelerator method 11.5.5, 11.8.4
setEnabled method 11.5.6, 11.8.4
setIcon method 11.5.2
JMOD files 9.1.1
JNDI service 5.10, 9.1.2
JndiLoginModule 9.2.1
JNI 13
JNICALL, JNIEXPORT macros 13.1
JobAttributes class (obsolete) 12.7.5

Join styles 12.5.5
Joining 1.9
joining method
of Collectors 1.13
JoinRowSet interface 5.7.1
JOptionPane class 11.8, 11.8.4
message types 11.8.1
showConfirmDialog method 11.8.1, 11.8.4
showInputDialog method 11.8.1, 11.8.4
showInternalConfirmDialog method 11.8.4
showInternalInputDialog method 11.8.4
showInternalMessageDialog method 11.8.4
showInternalOptionDialog method 11.8.4
showMessageDialog method 11.8.1, 11.8.4
showOptionDialog method 11.8.1, 11.8.4
JPanel class 11.2.1, 11.2.2, 12.7.1
JPasswordField class 11.8.4
Constructor 11.8.4
getPassword method 11.8.4
setEchoChar method 11.8.4
JPEG 12.6.1
image manipulations on 12.6.5
printing 12.7.3
reading 12.6.2
JPopupMenu class 11.5.4, 11.8.4
isPopupTrigger method 11.8.4
show method 11.8.4
JRadioButton class 11.4.2, 11.8.4
Constructor 11.8.4
JRadioButtonMenuItem class 11.8.4
Constructor 11.8.4
JRootPane class 11.8.4
setDefaultButton method 11.8.4
JRootPane class, setDefaultButton method 11.8.3
JScrollbar 10.4.7
JScrollPane class 11.8.4, 12.1.1
Constructor 11.8.4
JSlider class 11.4.5, 11.8.4
Constructor 11.8.4
setInverted method 11.4.5
setLabelTable method 11.4.5, 11.8.4
setMajorTickSpacing method 11.8.4
setMinorTickSpacing method 11.8.4
setPaintLabels method 11.8.4
setPaintLabels, setPaintTicks, setSnapToTicks methods 11.4.5
setPaintTicks method 11.8.4
setPaintTrack method 11.4.5, 11.8.4
setSnapToTicks method 11.8.4
JSP 8.1.6
Jstring 13.1

JTable class 12.1, 12.7.5
addColumn method 12.2.8, 12.7.5
Constructor 12.7.5
convertColumnIndexToModel method 12.7.5
convertRowIndexToModel method 12.7.5
convertXxxIndexToModel methods 12.2.5
default rendering actions 12.2.1
getAutoCreateRowSorter method 12.7.5
getCellRenderer method 12.3.1
getCellSelectionEnabled method 12.7.5
getColumnModel method 12.2.2, 12.7.5
getColumnSelectionAllowed method 12.7.5
getDefaultEditor method 12.7.5
getDefaultRenderer method 12.3.2, 12.7.5
getFillsViewportHeight method 12.7.5
getRowHeight method 12.7.5
getRowMargin method 12.7.5
getRowSelectionAllowed method 12.7.5
getSelectedColumns method 12.2.5, 12.2.8
getSelectedRows method 12.2.5
getSelectionModel method 12.7.5
getTableHeader method 12.1.1
installing cell editors automatically 12.3.3
moveColumn method 12.2.8, 12.7.5
print method 12.1.1, 12.7.5
removeColumn method 12.2.8, 12.7.5
resize modes of 12.2.3
setAutoCreateRowSorter method 12.1.1, 12.2.6, 12.7.5
setAutoResizeMode method 12.2.3, 12.7.5
setCellSelectionEnabled method 12.2.5, 12.7.5
setColumnSelectionAllowed method 12.2.5, 12.7.5
setDefaultRenderer method 12.3.1
setFillsViewportHeight method 12.7.5
setRowHeight method 12.7.5
setRowHeight, setRowMargin methods 12.2.4
setRowMargin method 12.7.5
setRowSelectionAllowed method 12.2.5, 12.7.5
setRowSorter method 12.2.6, 12.7.5
JTextArea class 11.3.4, 11.8.4
append method 11.8.4
Constructor 11.8.4
setColumns method 11.8.4
setColumns, setRows methods 11.3.4
setLineWrap method 11.3.4, 11.8.4
setRows method 11.8.4
setTabSize method 11.8.4
setWrapStyleWord method 11.8.4
JTextComponent
getText method 11.3.1
setText method 11.3.1

JTextComponent class 11.8.4
getText method 11.8.4
isEditable method 11.8.4
setEditable method 11.8.4
setText method 11.8.4
JTextField class 10.4.7, 11.3, 11.8.4, 12.3.3
Constructor 11.8.4
getColumns method 11.8.4
setColumns method 11.3.1, 11.8.4
JToolBar class 11.5.7, 11.8.4
add method 11.8.4
add, addSeparator methods 11.5.7
addSeparator method 11.8.4
Constructor 11.8.4
JTree class 12.4, 12.7.5
addTreeSelectionListener method 12.4.5
constructor 12.4.1, 12.7.5
getLastSelectedPathComponent method 12.4.2, 12.7.5
getSelectionPath method 12.4.2, 12.7.5
getSelectionPaths method 12.4.5, 12.7.5
identifying nodes 12.4.2
makeVisible method 12.4.2, 12.7.5
scrollPathToVisible method 12.4.2, 12.7.5
setEditable method 12.4.2
setRootVisible method 12.4.1, 12.7.5
setShowsRootHandles method 12.4.1, 12.7.5
JVM
class files in 9.1
class loaders in 9.1.1
specification for 9.1.5

K

Kerberos protocol 9.2.1
Kernel class 12.6.5, 12.7.5
Constructor 12.7.5
Kernel, of a convolution 12.6.5
Keyboard
associating with actions 10.4.5
focus of 10.4.5
mnemonics for 11.5.5
Keyboard, reading from 2.1.1
KeyEvent class 10.4.7
getKeyXxx, isActionKey methods 10.4.7
KeyGenerator class 9.4.2, 9.4.4
generateKey method 9.4.2, 9.4.4
getInstance method 9.4.4
init method 9.4.4
KeyListener interface, keyXxx methods 10.4.7
KeyPairGenerator class 9.4.4

keys method
of Preferences 10.5
KeyStoreLoginModule 9.2.1
Keystores 9.3.4
KeyStroke class 10.5
getKeyStroke method 10.5
KeyStroke class, getKeyStroke method 10.4.5
Keytool 9.3.4
Krb5LoginModule 9.2.1

L

L (object), type code 2.6.2
Labels
for components 11.3.2
for slider ticks 11.4.5
Lambda expressions 1.3
for event listeners 10.4.3
Landscape orientation 12.5.7
Language codes 1.9.3, 7.1.2
last method
of ResultSet 5.6.1, 5.10
lastAccessTime method
of BasicFileAttributes 2.7.7
lastDayOfMonth method
of TemporalAdjusters 6.7
lastDayOfYear method
of TemporalAdjusters 6.7
lastInMonth method
of TemporalAdjusters 6.7
lastModifiedTime method
of BasicFileAttributes 2.7.7
Layout algorithm 12.7.2
Layout management 11.2
border 11.2.2
box 11.6
custom 11.7
flow 11.2.1
grid 11.2.3
grid bag 11.6
group 11.6
sophisticated 11.6
spring 11.6
layoutContainer method
of LayoutManager 11.8.4
LayoutManager interface 11.8.4
addLayoutComponent method 11.8.4
designing custom 11.7
layoutContainer method 11.8.4
minimumLayoutSize method 11.8.4

preferredLayoutSize method 11.8.4
removeLayoutComponent method 11.8.4
LayoutManager2 interface 11.7
layoutPages method 12.7.2
Lazy operations 1.1, 1.2, 1.5, 2.7.5
LCD displays 12.6.4
LD_LIBRARY_PATH 13.1
Leading 10.3.3
Leap seconds 6.1
Leap years 6.2
Learn SQL The Hard Way (Shaw) 5.2
Learning SQL (Beaulieu) 5.2
Leaves 12.4, 12.4.1, 12.4.6
icons for 12.4.1, 12.4.4
Legacy APIs 2.3.1, 6.7
Legacy data, converting into XML 3.9
length method
of Blob 5.10
of CharSequence 2.7.7
of Clob 5.10
of RandomAccessFile 2.2.2, 2.7.7
lib/ext directory 9.1.1
libraryLookup method
of SymbolLookup 13.8.3
LIKE 5.2
LIKE (SQL) 5.5.3
limit method
of Buffer 2.7.7
of Stream 1.4, 1.13
Line feed 2.1.6, 7.7.2
in e-mails 4.6
in regular expressions 2.7.1
Line2D class 10.3.1, 12.5.2
Line2D.Double class 10.5
Constructor 10.5
Line2D.Double, Line2D.Float classes 12.5.2
LineBorder class 11.4.3, 11.8.4
LineBorder method 11.8.4
LineBorder method
of LineBorder 11.8.4
LineMetrics class 10.3.3, 10.5
getAscent method 10.5
getDescent method 10.5
getHeight method 10.5
getLeading method 10.5
Lines 10.3.1
constructing 10.3.1
lines method
of Files 1.2, 1.13, 2.3.2
of String 1.13

lineTo method 12.5.3
of Path2D.Float 12.7.5
Linker interface 13.2, 13.6, 13.8.3
canonicalLayouts method 13.8.3
defaultLookup method 13.8.3
downcallHandle method 13.8.3
nativeLinker method 13.8.3
upcallStub method 13.8.3
Linker.Option interface 13.8.3
captureCallState method 13.8.3
captureStateLayout method 13.8.3
firstVariadicArg method 13.8.3
Linux
library path in 13.1
pop-up menus in 11.5.4
using GNU C compiler 13.1
list method
of Files 2.3.6
Listener interfaces 10.4.1
Listener objects 10.4.1
ListResourceBundle class 7.8.3
Lists, converting to streams 1.13
ListSelectionModel interface 12.7.5
setSelectionMode method 12.2.5, 12.7.5
Little-endian order 2.1.8, 2.2.1, 2.4.1, 7.7.4
loadClass method
of ClassLoader 9.1.2, 9.1.4
of URLClassLoader 9.1.2
Loader lookup 13.6
loadLibrary method 13.1
of System 13.8.3
LOBs 5.5.2
placing in database 5.5.2
reading 5.5.2
Local hosts 4.1.4
Local names 3.6
LocalDate class 6.7, 7.9
datesUntil method 6.2, 6.7
format method 7.3
getDayOfMonth method 6.7
getDayOfWeek method 6.7
getDayOfXxx methods 6.2
getDayOfYear method 6.7
getMonth method 6.7
getMonthValue method 6.7
getYear method 6.7
isAfter method 6.7
isBefore method 6.7
isLeapYear method 6.7
legacy classes and 6.7

minus method 6.7
minus, minusXxx methods 6.2
minus*Xxx* method 6.7
now method 6.2, 6.7
of method 6.2, 6.3, 6.7
parse method 6.7, 7.3, 7.9
plus method 6.7
plus, plusXxx methods 6.2
plus*Xxx* method 6.7
toLocalDate method 6.7
until method 6.2, 6.7
weekends in 6.2
with method 6.3, 6.7
withDayOfMonth method 6.7
withDayOfYear method 6.7
withMonth method 6.7
withYear method 6.7
LocalDateTime class 6.4, 7.9
atZone method 6.5
format method 7.3
legacy classes and 6.7
parse method 7.3, 7.9
toLocalDateTime method 6.7
Locale class 7.1.2, 7.9
availableLocales method 1.9.2
forLanguageTag method 7.1.2, 7.9
getCountry method 1.9.3, 7.9
getDefault method 7.1.3, 7.9
getDisplayCountry method 7.9
getDisplayCountry, getDisplayLanguage methods 1.9.2
getDisplayLanguage method 7.9
getDisplayName method 7.1.4, 7.2.1, 7.9
getInstance method 7.4
getISOCountries method 7.1.2, 7.9
getISOLanguages method 7.1.2
getLanguage method 7.9
Locale.of method 7.9
predefined objects 7.1.2
setDefault method 7.1.3, 7.9
toLanguageTag method 7.1.2, 7.9
toString method 7.9
Locale.of method
of Locale 7.9
Locales 7.1
current 7.5.1
date and time formatting in 7.3
default 6.6, 7.1.3
dictionary ordering in 7.4
display names of 1.9.2, 7.1.4
formatting styles for 6.6

grouping 1.9.3
mapping names of 1.9.2
numbers in 7.1.1, 7.2
predefined 7.1.2
resources bundles and 7.8.1
variants in 7.1.2, 7.8.1
LocalTime class 6.4, 6.7, 7.9
method 6.7
format method 7.3
getHour method 6.7
getMinute method 6.7
getNano method 6.7
getSecond method 6.7
isAfter method 6.7
isBefore method 6.7
legacy classes and 6.7
minus method 6.7
minus, minusXxx methods 6.4
now method 6.4, 6.7
of method 6.4, 6.7
parse method 7.3, 7.9
plus method 6.7
plus, plusXxx methods 6.4
toLocalTime method 6.7
toNanoOfDay method 6.7
toSecondOfDay method 6.7
with*Xxx* method 6.7
lock method
of FileChannel 2.5, 2.7.7
Locks 2.5
for the tail portion of a file 2.5
shared 2.5
unlocking 2.5
logging.properties file 4.4.4
login method
of LoginContext 9.4.4
of LoginModule 9.4.4
LoginContext class 9.2.1, 9.4.4
Constructor 9.4.4
getSubject method 9.4.4
login method 9.4.4
login, logout methods 9.2.1
logout method 9.4.4
LoginModule interface 9.4.4
abort method 9.4.4
commit method 9.4.4
documentation for 9.2.2
initialize method 9.4.4
login method 9.4.4
logout method 9.4.4

Logins
committed 9.2.2
modules for 9.2.1
custom 9.2.2
separating from action code 9.2.2
logout method
of LoginContext 9.4.4
of LoginModule 9.4.4
Long class
MAX_VALUE constant 2.5
LONG NVARCHAR 5.9.4
long type
printing 2.1.6
streams of 1.12
type code for 2.6.2
writing in binary format 2.2.1
LONG VARCHAR 5.9.4
LongBuffer class 2.4.2
longs method
of Random 1.12
of RandomGenerator 1.13
LongStream interface 1.12, 1.13
average method 1.12, 1.13
boxed method 1.12, 1.13
mapToLong method 1.12
max method 1.13
max, min methods 1.12
min method 1.13
of method 1.13
range method 1.13
range, rangeClosed methods 1.12
rangeClosed method 1.13
sum method 1.13
sum, summaryStatistics methods 1.12
summaryStatistics method 1.13
toArray method 1.12, 1.13
LongSummaryStatistics class 1.9, 1.12, 1.13
getAverage method 1.13
getCount method 1.13
getMax method 1.13
getMin method 1.13
getSum method 1.13
Look-and-feel
appearance of buttons in 11.1
displaying trees in 12.4.1
handles for subtrees in 12.4.4
pluggable 11.8.4
selecting multiple nodes in 12.4.5
lookingAt method
of Matcher 2.7.7

lookup method
of MethodHandles 13.8.3
LookupOp class 12.6.5, 12.7.5
Constructor 12.7.5
lookupPrintServices method 12.7.3
of PrintServiceLookup 12.7.5
lookupStreamPrintServiceFactories method
of StreamPrintServiceFactory 12.7.5
LookupTable class 12.6.5
Low-level events 10.4.7
LSB (least significant byte) 2.2.1
LSOutput 3.8.3
LSSerializer 3.8.3
writeToString method 3.8.3

M

Mac OS X
character encodings in 7.7
resources in 7.8
Mac Roman 7.7
main method 9.1.1
makeShape method 12.5.3
makeVisible method 12.4.2
of JTree 12.7.5
Mandelbrot set 12.6.4
Mangling names 13.1
Manifest files 2.2.3
map method
of FileChannel 2.4.1, 2.7.7
of HttpHeaders 4.4.3, 4.6
of Optional 1.7.3, 1.13
of Stream 1.3, 1.13
mapConcurrent method
of Gatherers 1.13
mapMulti method
of Stream 1.13
mapping method
of Collectors 1.9.4, 1.13
Maps
concurrent 1.9.2
of stream elements 1.9.2, 1.13
mapToInt method 1.10
mapTo*Xxx* methods
of *Xxx*Stream 1.12
mark method
of Buffer 2.4.2, 2.7.7
of InputStream 2.7.7
markSupported method
of InputStream 2.7.7

Mastering Regular Expressions (Friedl) 2.7.1
match attribute (XSLT) 3.9
Matcher class 2.7.7
find method 2.7.7
hasMatch method 2.7.7
lookingAt method 2.7.7
matches method 2.7.7
of Pattern 2.7.2
quoteReplacement method 2.7.7
replaceAll method 2.7.6, 2.7.7
replaceFirst method 2.7.6, 2.7.7
results method 2.7.7
matcher method
of Pattern 2.7.2, 2.7.7
matches method
of Matcher 2.7.7
of PathMatcher 2.7.7
of Pattern 2.7.2
MatchResult interface 2.7.3, 2.7.7
end method 2.7.7
group method 2.7.7
namedGroups method 2.7.7
start method 2.7.7
Matisse 11.6
Matrices, transformations of 12.5.7
Max
of primitive streams 1.12
max method
of DoubleStream 1.13
of IntStream 1.13
of LongStream 1.13
of Stream 1.6, 1.13
MAX_VALUE 2.5
maxBy method 1.9.4
of Collectors 1.13
maxoccurs attribute (XSLT) 3.4.2
MD5 9.3.1
Memory addresses, vs. serial numbers 2.6.1
Memory layout 13.5
hierarchy 13.5
Memory layouts
canonical 13.5
Memory mapping of files 2.4
Memory segment 13.4
allocating 13.4
off-heap 13.3, 13.4
on-heap 13.4
reading 13.4
writing 13.4
MemoryLayout interface 13.8.3

byteOffset method 13.8.3
byteOffsetHandle method 13.8.3
sequenceLayout method 13.8.3
structLayout method 13.8.3
unionLayout method 13.8.3
varHandle method 13.8.3
MemoryLayout.PathElement interface 13.8.3
dereferenceElement method 13.8.3
groupElement method 13.8.3
sequenceElement method 13.8.3
MemorySegment interface 13.8.3
get method 13.8.3
getAtIndex method 13.8.3
getString method 13.8.3
reinterpret method 13.8.3
set method 13.8.3
setAtIndex method 13.8.3
toArray method 13.8.3
menuCanceled method
of MenuListener 11.8.4
menuDeselected method
of MenuListener 11.8.4
MenuListener interface 11.5.6, 11.8.4
menuCanceled method 11.8.4
menuDeselected method 11.8.4
menuSelected method 11.8.4
menuXxx methods 11.5.6
Menus 11.5
accelerators for 11.5.5
checkboxes in 11.5.3
icons in 11.5.2
keyboard mnemonics for 11.5.5
menu bar in 11.5, 11.5.1
menu items in 11.5
enabling/disabling 11.5.6
pop-up 11.5.4
radio buttons in 11.5.3
submenus in 11.5, 11.5.1
menuSelected method
of MenuListener 11.8.4
Message digests 9.3.1
MessageDigest class 9.4.4
digest method 9.3.1, 9.4.4
extending 9.3.1
getInstance method 9.3.1, 9.4.4
reset method 9.4.4
update method 9.3.1, 9.4.4
MessageFormat class 7.5, 7.9
applyPattern method 7.9
Constructor 7.9

format method 7.5.1, 7.9
getLocale method 7.9
ignoring the first limit 7.5.2
setLocale method 7.9
Metadata 5.8
Metal look-and-feel
selecting multiple nodes in 12.4.5
trees in 12.4.1
Method handle 13.6
downcall 13.6
Method verification errors 9.1.5
MethodHandle class 13.2, 13.8.3
invoke method 13.8.3
MethodHandle.Lookup 13.8.3
findStatic method 13.8.3
MethodHandles class 13.8.3
lookup method 13.8.3
Methods
do-nothing 10.4.4
mangling names of 13.1
native 13
protected 9.1.2
MethodType class 13.8.3
methodType method 13.8.3
methodType method
of MethodType 13.8.3
Microsoft
compiler 13.1
ODBC API 5
SQL Server 5.3
Visual Basic 10.4
Microsoft Windows. See Windows operating system
MIME 12.6.2
for print services 12.7.3
getting, of a file 2.3.2
MimeMessage class
methods of 4.6
Min
of primitive streams 1.12
min method
of DoubleStream 1.13
of IntStream 1.13
of LongStream 1.13
of Stream 1.6, 1.13
minBy method 1.9.4
of Collectors 1.13
minimumLayoutSize method
of LayoutManager 11.8.4
minoccurs attribute (XSLT) 3.4.2
minus method

of Duration 6.7
of Instant 6.7
of LocalDate 6.7
of LocalTime 6.7
of Period 6.7
of ZonedDateTime 6.7
minus, minusXxx methods
of LocalDate 6.2
of LocalTime 6.4
minus*Xxx* method
of Duration 6.7
of Instant 6.7
of LocalDate 6.7
of Period 6.7
of ZonedDateTime 6.7
mismatch method
of Files 2.3.2, 2.7.7
MissingResourceException class 7.8.1
Miter join, miter limit 12.5.5
Mixed content 3.2, 3.4.1
Modality 11.8, 11.8.2
Model-view-controller 11.1
classes in 11.1
multiple views in 11.1
Modernist painting example 3.8.4
Modified UTF-8 2.2.1, 7.7.5
native code and 13.1
Modules
platform classes in 9.1.1
Month enumeration 6.2
getDisplayName method 6.6, 7.3
MonthDay class 6.2
Mouse events 10.4.6
MouseAdapter class 10.4.6
MouseEvent class 10.4.7, 10.5, 11.8.4
getClickCount method 10.4.6, 10.4.7, 10.5
getPoint method 10.4.7, 10.5
getX method 10.5
getX/Y methods 10.4.6, 10.4.7
getY method 10.5
isPopupTrigger method 11.8.4
translatePoint method 10.4.7
MouseHandler 10.4.6
MouseListener interface 10.4.6
mouseClicked method 10.4.6, 10.4.7
mouseDragged method 10.4.6
mouseEntered/Exited methods 10.4.6, 10.4.7
mousePressed/Released methods 10.4.6, 10.4.7
MouseMotionHandler 10.4.6
MouseMotionListener interface 10.4.6

mouseDragged method 10.4.7
mouseMoved method 10.4.6, 10.4.7
MouseWheelEvent class 10.4.7
getScrollAmount, getWheelRotation methods 10.4.7
MouseWheelListener interface, mouseWheelMoved method 10.4.7
move method
of Files 2.3.4, 2.7.7
moveColumn method 12.2.8
of JTable 12.7.5
moveTo method 12.5.3
of Path2D.Float 12.7.5
moveToCurrentRow method
of ResultSet 5.10
moveToInsertRow method
of ResultSet 5.10
moveToXxxRow methods (ResultSet) 5.6.2
MSB (most significant byte) 2.2.1
Multiple-page printing 12.7.2
multipliedBy method
of Duration 6.7
MutableTreeNode interface 12.7.5
implementing 12.4.1
setUserObject method 12.4.1, 12.7.5
MySQL 5.3

N

NameCallback class 9.2.2, 9.4.4
Constructor 9.4.4
getDefaultName method 9.4.4
getName method 9.4.4
getPrompt method 9.4.4
setName method 9.4.4
namedGroups method
of MatchResult 2.7.7
of Pattern 2.7.7
NamedNodeMap interface 3.9
getLength method 3.9
item method 3.9
Namespaces 3.6
activating processing of 3.4.2
aliases (prefixes) for 3.4.2, 3.6
of attributes 3.6
of child elements 3.6
using class loaders as 9.1.3
National character strings 5.9.4
National Institute of Standards and Technology 4.1.1, 9.3.1
native keyword 13.1
Native methods 13
calling from Java programs 13.1

compiling 13.1
garbage collection and 13.1
linking to Java 13.1
naming 13.1
overloading 13.1
reasons to use 13
static 13.1
strings in 13.1
native.encoding package 2.1.9
nativeLinker method
of Linker 13.8.3
NCHAR, NCLOB data types (SQL) 5.9.4
negated method
of Duration 6.7
Nervous text applet 9.3
net.properties package 4.4.4
NetBeans
Matisse 11.6
Netscape
IFC library 10.1
Networking 4, 4.6
connecting to a server 4.1
debugging 4.1.1
getting web data 4.3
HTTP client for 4.4
implementing servers 4.2
sending e-mails 4.6
newBufferedReader method
of Files 2.7.7
newBufferedWriter method
of Files 2.7.7
newBuffered*Xxx* methods
of Files 2.3.2
newBuilder method
of HttpClient 4.4.1, 4.6
of HttpRequest 4.4.4, 4.6
newDirectoryStream method 2.3.8
of Files 2.7.7
newDocument method 3.8.1, 3.9
of DocumentBuilder 3.9
newDocumentBuilder method 3.3, 3.8.2
of DocumentBuilderFactory 3.9
newFactory method 5.7.1
of RowSetProvider 5.10
newFileSystem method 2.3.9
of FileSystems 2.7.7
newHttpClient method 4.4.1
of HttpClient 4.6
newInputStream method
of Channels 4.6

of Files 2.3.2, 2.7.7
newInstance method
of DocumentBuilderFactory 3.3, 3.6, 3.9
of SAXParserFactory 3.7.1, 3.9
of TransformerFactory 3.8.3, 3.9
of XMLInputFactory 3.9
of XMLOutputFactory 3.8.4, 3.9
of XPathFactory 3.5, 3.9
newNSInstance method 3.6, 3.7.1
of DocumentBuilderFactory 3.9
of SAXParserFactory 3.9
newOutputStream method
of Channels 4.2.4, 4.6
of Files 2.3.2, 2.7.7
newSAXParser method 3.7.1
of SAXParserFactory 3.9
NewStringUTF 13.1
newTransformer method 3.8.3
of TransformerFactory 3.9
newXpath method
of XPathFactory 3.9
next method
of ResultSet 5.4.1, 5.6, 5.10
of TemporalAdjusters 6.7
of XMLStreamReader 3.9
nextElement method 12.4.3
nextOrSame method 6.3
of TemporalAdjusters 6.7
nextPage method 5.7.2
of CachedRowSet 5.10
NMTOKEN, NMTOKENS attribute types (DTDs) 3.4.1
No-argument constructors 2.6.1, 2.6.8
Node interface 3.9
appendChild method 3.8.1, 3.9
getAttributes method 3.3, 3.9
getChildNodes method 3.3, 3.9
getFirstChild method 3.3, 3.9
getLastChild method 3.3, 3.9
getLocalName method 3.6, 3.9
getNamespaceURI method 3.6, 3.9
getNextSibling method 3.3, 3.9
getNodeName method 3.9
getNodeValue method 3.9
getNodeXxx methods 3.3, 3.6
getParentNode method 3.9
getPreviousSibling method 3.9
subinterfaces of 3.3
node method
of Preferences 10.5
nodeChanged method 12.4.2

of DefaultTreeModel 12.7.5
NodeList interface 3.3, 3.9
getLength method 3.3, 3.9
item method 3.3, 3.4.1, 3.9
Nodes class 12.4
adding 12.4.2
child 12.4, 12.4.1
collapsed 12.4.2
connecting lines for 12.4.1
currently selected 12.4.2
editing 12.4.2, 12.4.6
enumerating 12.4.3
expanding 12.4.2
handles for 12.4.1, 12.4.4
highlighting 12.4.4
identifying, by tree paths 12.4.2
making visible 12.4.2
parent 12.4, 12.4.1
removing 12.4.2
rendering 12.4.1, 12.4.4
root 12.4, 12.4.1
row positions of 12.4.2
searching 12.4.3, 12.4.5
selecting 12.4.5
user objects for 12.4.1, 12.4.2, 12.4.3, 12.4.5
nodesChanged method
of DefaultTreeModel 12.7.5
Nondeterministic parsing 3.4.1
noneMatch method 1.6
of Stream 1.13
Noninterference, of stream operations 1.2
Normalization 7.4
normalize method
of Normalizer 7.9
of Path 2.3.1, 2.7.7
Normalized color values 12.6.4
Normalizer class 7.4, 7.9
normalize method 7.9
NoSuchElementException class 1.7.4
notFilter method 12.2.7
of RowFilter 12.7.5
NotSerializableException class 2.6.4
now method
of Instant 6.1, 6.7
of LocalDate 6.2, 6.7
of LocalTime 6.4, 6.7
of ZonedDateTime 6.7
NTLoginModule 9.2.1
nullInputStream method
of InputStream 2.7.7

nullOutputStream method
of OutputStream 2.7.7
NullPointerException class
vs. Optional 1.6
Number class
doubleValue method 7.2.1
intValue method 7.2.1
numberFilter method 12.2.7
of RowFilter 12.7.5
NumberFormat class 7.2, 7.9
format method 7.2.1, 7.9
getAvailableLocales method 7.1.2, 7.2.1, 7.9
getCompactNumberInstance method 7.9
getCurrencyInstance method 7.2.1, 7.2.3, 7.9
getMaximumFractionDigits method 7.9
getMaximumIntegerDigits method 7.9
getMinimumFractionDigits method 7.9
getMinimumIntegerDigits method 7.9
getNumberInstance method 7.2.1, 7.9
getPercentInstance method 7.2.1, 7.9
isGroupingUsed method 7.9
isParseIntegerOnly method 7.9
parse method 7.2.1, 7.9
setCurrency method 7.2.3
setGroupingUsed method 7.9
setMaximumFractionDigits method 7.9
setMaximumIntegerDigits method 7.9
setMinimumFractionDigits method 7.9
setMinimumIntegerDigits method 7.9
setParseIntegerOnly method 7.9
Numbers
filtering 12.2.7
floating-point 7.1.1, 7.2
formatting 7.1.1, 7.2, 7.5.1
supported locales for 7.2.1
from grouped elements 1.9.4
in regular expressions 2.7.1
printing 2.1.6
random 1.2, 1.4, 1.12, 9.4.2
reading
from files 2.1.3
from ZIP archives 2.1.3
using locales 7.2.1
storing in memory 2.2.1
writing in binary format 2.2.1
NUMERIC 5.2, 5.9.4
NVARCHAR 5.9.4

O

Object class
clone method 2.6.1, 2.6.9
Object inspection tree 12.4.6
Object serialization 2.6
cloning with 2.6.9
file format for 2.6.2
serial numbers for 2.6.1
ObjectInputStream class 2.6.1, 2.7.7
Constructor 2.7.7
readObject method 2.6.1, 2.6.4, 2.7.7
ObjectInputValidation interface 2.6.8
ObjectOutputStream class 2.6.1, 2.7.7
Constructor 2.7.7
defaultWriteObject method 2.6.4
writeObject method 2.6.1, 2.6.4, 2.7.7
Objects
cloning 2.6.9
converting to streams 1.2, 1.7.7
deserialized 2.6.6
fingerprints of 2.6.2
printing 2.1.6
reading from an input stream 2.6.1
saving
in output streams 2.6.1
in text format 2.1.7
serializable 2.6.1
transmitting over network 2.6.1
type code for 2.6.2
versioning 2.6.7
ODBC 5, 5.1.1
of method
of Collector 1.13
of DoubleStream 1.13
of FunctionDescriptor 13.8.3
of Gatherer 1.13
of Gatherer.Integrator 1.13
of IntStream 1.12, 1.13
of LocalDate 6.2, 6.3, 6.7
of LocalTime 6.4, 6.7
of LongStream 1.13
of Optional 1.7.5, 1.13
of OptionalDouble 1.13
of OptionalInt 1.13
of OptionalLong 1.13
of Path 2.3.1, 2.7.7
of Period 6.7
of Stream 1.2, 1.13
of ZonedDateTime 6.5, 6.7
of ZoneId 6.5
ofAuto method

of Arena 13.8.3
ofConfined method
of Arena 13.8.3
ofDateAdjuster method 6.3
ofDays method
of Period 6.2, 6.5
Off-heap memory segment 13.3
ofFile, ofFileDownload methods
of BodyHandlers 4.4.3
OffsetDateTime class 6.5
ofInstant method
of ZonedDateTime 6.7
ofLocalizedDate method
of DateTimeFormatter 6.7, 7.9
ofLocalizedDateTime method
of DateTimeFormatter 6.7, 7.9
ofLocalizedPattern method
of DateTimeFormatter 6.7
ofLocalizedTime method
of DateTimeFormatter 6.7, 7.9
ofLocalizedXxx methods
of DateTimeFormatter 6.6, 7.3
ofMonths method 6.2
ofNullable method
of Optional 1.7.5, 1.13
of Stream 1.2, 1.7.7, 1.13
ofPattern method 6.6
of DateTimeFormatter 6.7
ofSequential method
of Gatherer 1.13
ofShared method
of Arena 13.8.3
ofString method
of BodyHandlers, BodyPublishers 4.4.2, 4.4.4
ofVoid method
of FunctionDescriptor 13.8.3
of*Xxx* method
of Duration 6.7
of Period 6.7
oj (SQL escapes) 5.5.3
open keyword
of FileChannel 2.4.1
of SocketChannel 4.2.4
open method
of FileChannel 2.7.7
of SocketChannel 4.6
Open path element 13.8.1
openConnection method 4.3.2
of URL 4.6
openOutputStream method

of SimpleJavaFileObject 8.2.6
openStream method 2.1.3, 4.3.1
of URL 4.6
Operating system
character encodings in 2.1.9, 7.7
default locales in 7.1.3
paths in 2.1.3, 2.3.1
resources in 7.8
Operations
associative 1.10
lazy 1.1, 1.2, 1.5, 2.7.5
stateless 1.13
Option panes 11.8.1
Optional class 1.6, 1.7.6, 1.13
creating values of 1.7.5
empty method 1.7.5, 1.13
filter method 1.7.3, 1.13
flatMap method 1.7.6, 1.13
for empty streams 1.10
get method 1.7.4, 1.7.7, 1.13
ifPresent method 1.7.2, 1.13
ifPresentOrElse method 1.7.2, 1.13
isEmpty method 1.13
isPresent method 1.7.4, 1.7.7, 1.13
map method 1.7.3, 1.13
of method 1.13
of, ofNullable methods 1.7.5
ofNullable method 1.13
or method 1.7.3, 1.13
orElse method 1.6, 1.13
orElseGet method 1.13
orElseThrow method 1.13
stream method 1.7.7
optional option (JAAS configuration) 9.2.1
OptionalDouble class 1.13
getAs*Xxx* method 1.13
ifPresent method 1.13
of method 1.13
orElse method 1.13
orElseGet method 1.13
OptionalInt class 1.13
getAs*Xxx* method 1.13
ifPresent method 1.13
of method 1.13
orElse method 1.13
orElseGet method 1.13
OptionalInt, OptionalDouble, OptionalLong classes 1.12
OptionalLong class 1.13
getAs*Xxx* method 1.13
ifPresent method 1.13

of method 1.13
orElse method 1.13
orElseGet method 1.13
or method
of Optional 1.13
Oracle
JDBC 5, 5.3
JVM implementation 2.1.9
ORDER BY 5.4.1
order method
of ByteBuffer 2.4.1, 2.7.7
orElse method
of Optional 1.13
of OptionalDouble 1.13
of OptionalInt 1.13
of OptionalLong 1.13
orElseGet method
of Optional 1.13
of OptionalDouble 1.13
of OptionalInt 1.13
of OptionalLong 1.13
orElseThrow method
of Optional 1.13
orFilter method 12.2.7
of RowFilter 12.7.5
org.w3c.dom package 3.3
Outer joins 5.5.3
output element (XSLT) 3.9
Output streams 2.1
buffered 2.1.3
byte processing in 2.1.3
byte-oriented 2.1
closing 2.1.1, 4.2.3
filters for 2.1.3
hierarchy of 2.1.2
objects in 2.6
redirecting 8.2.3
Unicode and 2.1
OutputStream class 2.1, 2.1.2, 2.7.7, 3.8.4
close method 2.7.7
flush method 2.1.1, 2.7.7
nullOutputStream method 2.7.7
write method 2.1.1, 2.7.7
OutputStreamWriter class 2.1.4
OverlappingFileLockException class 2.5
Overloading 13.1
Owner frame 11.8.2
Owner thread
of arena 13.3

P

pack method
of Window 10.3, 10.5
Packages
avoiding name clashes with 9.1.3
Packets 4.1.2
Padding schemes 9.4.1
PaddingLayout interface 13.5
Page setup dialog box 12.7.1
Pageable interface
implementing 12.7.2
objects, printing 12.7.3
PageAttributes class (obsolete) 12.7.5
pageDialog method 12.7.1
of PrinterJob 12.7.5
PageFormat class 12.7.5
getHeight method 12.7.1, 12.7.5
getImageableHeight method 12.7.5
getImageableWidth method 12.7.5
getImageableX method 12.7.5
getImageableXxx methods 12.7.1
getImageableY method 12.7.5
getOrientation method 12.7.5
getWidth method 12.7.1, 12.7.5
Pages
measurements of 12.7.1
multiple, printing 12.7.2
orientation of 12.5.7, 12.7.1
Paint interface 12.5.6
paint method
of JComponent 12.3.1, 12.5.1
paintComponent method 10.3, 10.3.3, 10.3.4
of JComponent 10.5, 12.5.1
of StrokePanel 12.5.5
overriding 10.4.7
Paper class
margins of 12.7.1
sizes of 7.8.3, 12.7.1
parallel method
of BaseStream 1.13
parallelStream method 1.1, 1.13
of Collection 1.13
ParameterMetaData interface 5.10
getParameterClassName method 5.10
getParameterCount method 5.10
getParameterMode method 5.10
getParameterType method 5.10
getParameterTypeName method 5.10
getPrecision method 5.10

getScale method 5.10
isNullable method 5.10
Parent classes. See Superclasses
Parent nodes 12.4, 12.4.1
parse method
of DateTimeFormatter 6.6
of DocumentBuilder 3.9
of LocalDate 6.7, 7.3, 7.9
of LocalDateTime 7.9
of LocalTime 7.9
of NumberFormat 7.2.1, 7.9
of SAXParser 3.7.1, 3.9
of XMLReader 3.9
of ZonedDateTime 6.7, 7.3, 7.9
Parsed character data 3.4.1
ParseException class 7.2.1
Parsers 3.3
checking uniqueness of IDs in 3.4.1, 3.4.3
pull 3.7.2
validating in 3.4
Parsing 3.3
nondeterministic 3.4.1
with XML Schema 3.4.2
partitioningBy method 1.9.3, 1.9.4
of Collectors 1.13
Password-protected resources 4.3.2
PasswordCallback class 9.2.2, 9.4.4
Constructor 9.4.4
getPassword method 9.4.4
getPrompt method 9.4.4
isEchoOn method 9.4.4
setPassword method 9.4.4
PasswordChooser 11.8.3
Passwords
dialog box for 11.8.3
fields for 11.3.3
Path element
open 13.8.1
Path interface 2.3, 2.3.1, 2.7.7
getFileName method 2.7.7
getParent method 2.7.7
getRoot method 2.7.7
getXxx methods 2.3.1
normalize method 2.3.1, 2.7.7
of method 2.3.1, 2.7.7
relativize method 2.3.1, 2.7.7
resolve method 2.7.7
resolve, resolveSibling methods 2.3.1
resolveSibling method 2.7.7
toAbsolutePath method 2.7.7

toAbsolutePath, toFile methods 2.3.1
toFile method 2.7.7
Path2D class 12.7.5
append method 12.5.3, 12.5.8, 12.7.5
closePath method 12.5.3, 12.7.5
Path2D.Double class 12.5.2
Path2D.Float class 12.5.2, 12.7.5
curveTo method 12.7.5
curveTo, lineTo, moveTo, quadTo methods 12.5.3
lineTo method 12.7.5
moveTo method 12.7.5
quadTo method 12.7.5
pathFromAncestorEnumeration method 12.4.3
PathMatcher interface 2.7.7
matches method 2.7.7
Paths (file system) 2.3.1
absolute vs. relative 2.1.3, 2.3.1
checking properties of 2.3.5
filtering 2.3.6
relativizing 2.3.1
resolving 2.3.1
root component of 2.3.1
separators in 2.1.3, 2.3.1
Paths (graphics) 12.5.3
Paths class 2.3.9
get method 2.3.1
Pattern class 1.13, 2.7.1, 2.7.7
asMatchPredicate, asPredicate methods 2.7.2
compile method 2.7.2, 2.7.7
flags 2.7.7
matcher method 2.7.2, 2.7.7
matches method 2.7.2
namedGroups method 2.7.7
split method 2.7.5, 2.7.7
splitAsStream method 1.2, 1.13, 2.7.5, 2.7.7
splitWithDelimiters method 2.7.7
Patterns 2.7
#PCDATA 3.4.1
PDF format 12.7.3
peek method
of Stream 1.5, 1.13
Percentages, formatting 7.2, 7.2.1
Performance
of encryption algorithms 9.4.4
of file operations 2.4.1
Period class 6.7
get*Xxx* method 6.7
minus method 6.7
minus*Xxx* method 6.7
of method 6.7

ofXxx method 6.7
ofXxx methods 6.2, 6.5
plus method 6.7
plusXxx method 6.7
using for daylight savings time 6.5
withXxx method 6.7
Perl programming language 2.7.1
Personal data 9.4
Pixels
affine transformations on 12.6.5
average value of 12.6.5
composing 12.5.9
interpolating 12.5.6, 12.6.5
sample values of 12.6.4
setting individual 12.6.4
Placeholders, in message formatting 7.5.2
Placeholders, in messages 7.5.1
Plugins, loading 9.1.2
plus method
of Duration 6.7
of Instant 6.7
of LocalDate 6.7
of LocalTime 6.7
of Period 6.7
of ZonedDateTime 6.7
plus, plusXxx methods
of LocalDate 6.2
of LocalTime 6.4
plusXxx method
of Duration 6.7
of Instant 6.7
of LocalDate 6.7
of Period 6.7
of ZonedDateTime 6.7
PNG 12.6.1
printing 12.7.3
Point class 10.3.1
Point size 10.3.3
Point2D class 10.3.1, 12.5.2
Point2D.Double class 2.6.4, 10.3.1, 10.5, 12.5.2
Constructor 10.5
Point2D.Float class 10.3.1, 12.5.2
Points, in typography 12.7.1
Policy files 9.3.7
Polygons 12.5.2, 12.5.3
Pools, for parallel streams 1.13
Pop-up menus 11.5.4
populate method
of CachedRowSet 5.7.2, 5.10
Porter–Duff rules 12.5.9

Portrait orientation 12.7.1
Ports 4.1.1
blocking 4.1.1
in URIs 4.3.1
position function (XPath) 3.9
POSIX-compliant file systems 2.3.5
PosixFileAttributes interface 2.3.5
POST 4.3.3
building 4.4.2
POST method
of HttpRequest.Builder 4.4.2, 4.6
PostgreSQL 5.3
connecting to 5.3.4
drivers for 5.3.2
Postorder traversal 12.4.3
postOrderEnumeration method 12.4.3
of DefaultMutableTreeNode 12.7.5
PostScript
printing 12.7.3, 12.7.4
writing to 12.7.4
postVisitDirectory method
of FileVisitor 2.3.8
of SimpleFileVisitor 2.7.7
Predefined action table names 10.4.5
Predefined character classes 2.7.1
Predicate functions 1.9.3
Preferences
accessing 10.5
enumerating keys in 10.5
importing/exporting 10.5
Preferences class 10.5
exportNode method 10.5
exportSubtree method 10.5
export*Xxx* methods 10.5
get method 10.5
get, get*DataType* methods 10.5
getBoolean method 10.5
getByteArray method 10.5
getDouble method 10.5
getFloat method 10.5
getInt method 10.5
getLong method 10.5
importPreferences method 10.5
keys method 10.5
node method 10.5
platform-independency of 10.5
put method 10.5
putBoolean method 10.5
putByteArray method 10.5
putDouble method 10.5

putFloat method 10.5
putInt method 10.5
putLong method 10.5
system/userNodeForPackage methods 10.5
system/userRoot methods 10.5
systemNodeForPackage method 10.5
systemRoot method 10.5
userNodeForPackage method 10.5
userRoot method 10.5
preferredLayoutSize method
of LayoutManager 11.8.4
preOrderEnumeration method 12.4.3
of DefaultMutableTreeNode 12.7.5
Prepared statements 5.5.1
caching 5.5.1
executing 5.5.1
PreparedStatement interface 5.10
clearParameters method 5.10
executeQuery method 5.10
executeUpdate method 5.10
execute*Xxx* methods 5.5.1
getMetaData method 5.10
set*Xxx* method 5.10
set*Xxx* methods 5.5.1
prepareStatement method 5.5.1, 5.6.1
of Connection 5.10
Previous 5.6.1
previous method
of ResultSet 5.6.1, 5.10
of TemporalAdjusters 6.7
previousOrSame method
of TemporalAdjusters 6.7
previousPage method
of CachedRowSet 5.10
preVisitDirectory method
of FileVisitor 2.3.8
of SimpleFileVisitor 2.7.7
Primitive types
I/O in binary format in 2.1.2
streams of 1.10
Principal interface 9.4.4
getName method 9.4.4
Principals (logins) 9.2.1
Print dialog box 12.7.1
displaying page ranges in 12.7.1, 12.7.2
native 12.7.1
print method
of DocPrintJob 12.7.5
of JTable 12.1.1, 12.7.5
of Printable 12.7.1, 12.7.2, 12.7.5

of PrinterJob 12.7.5
of PrintWriter 2.1.6, 2.7.7
Print services 12.7.3
document flavors for 12.7.3
for images 12.7.3
stream 12.7.4
print, println functions (JavaScript) 8.2.3
Printable interface 12.7.5
implementing 12.7.1
objects, printing 12.7.3
print method 12.7.1, 12.7.2, 12.7.5
printDialog method
of PrinterJob 12.7.5
Printer graphics context 12.7.2
PrinterException class 12.7.1
PrinterJob class 12.7.5
defaultPage method 12.7.5
getPrinterJob method 12.7.1, 12.7.5
pageDialog method 12.7.1, 12.7.5
print method 12.7.1, 12.7.5
printDialog method 12.7.1, 12.7.5
setPageable method 12.7.2
setPrintable method 12.7.5
printf method
of PrintWriter 2.1.6, 2.7.7, 7.1.4
Printing
clipped areas 12.7.1
counting pages during 12.7.1
images 12.7.1
layout of 12.7.2
multipage documents 12.7.2
number of copies for 12.7.5
page orientation of 12.5.7, 12.7.1
paper sizes in 12.7.1
quality of 12.7.5
selecting settings for 12.7.1
starting 12.7.1
text 12.7.1
using
banding for 12.7.1
transformations for 12.7.2
Printing attributes 12.7.5
adding 12.7.5
categories of 12.7.5
checking values of 12.7.5
hierarchy of 12.7.5
retrieving 12.7.5
PrintJob class (obsolete) 12.7.1
PrintJobAttribute interface 12.7.5
printing attributes of 12.7.5

PrintJobAttributeSet interface 12.7.5
println method
of PrintWriter 2.1.6, 2.1.7, 2.7.7
of System.out 7.7.2, 7.7.3
PrintQuality 12.7.5
PrintRequestAttribute interface 12.7.5
printing attributes of 12.7.5
PrintRequestAttributeSet interface 12.7.1, 12.7.5
PrintService interface 12.7.5
createPrintJob method 12.7.3, 12.7.5
getAttributes method 12.7.5
getName method 12.7.3
PrintServiceAttribute interface 12.7.5
printing attributes of 12.7.5
PrintServiceAttributeSet interface 12.7.5
PrintServiceLookup class 12.7.5
lookupPrintServices method 12.7.3, 12.7.5
PrintStream class 2.1.6
PrintWriter class 2.1.6, 2.7.7
checkError method 2.1.6, 2.7.7
Constructor 2.7.7
print method 2.1.6, 2.7.7
printf method 2.1.6, 2.7.7, 7.1.4
println method 2.1.6, 2.1.7, 2.7.7
Private keys 9.3.3, 9.4.4
probeContentType method 2.3.2
of Files 2.7.7
Processing instructions 3.2
Programmer's Day 6.2
Programs. See Applications
Project Panama 13.2
Properties class 3.1, 10.2.2
Property files 3.1
character encoding of 7.8.2
for resources bundles 7.8.1
for string resources 7.8
for strings 7.8.2
no passwords in 4.6
PropertyChangeListener interface 11.8.4
PUBLIC 3.8.3
Public key ciphers 9.3.3, 9.3.5, 9.4.4
performance of 9.4.4
Public Key Cryptography Standard (PKCS) #5 9.4.1
Pull parsers 3.7.2
push method
of Gatherer.Downstream 1.13
PushbackInputStream class 2.1.3, 2.7.7
Constructor 2.7.7
unread method 2.7.7
put method

of Bindings 8.2.6
of ByteBuffer 2.7.7
of CharBuffer 2.7.7
of HttpRequest.Builder 4.6
of Preferences 10.5
of ScriptEngine 8.2.6
of ScriptEngineManager 8.2.6
putBoolean method
of Preferences 10.5
putByteArray method
of Preferences 10.5
putClientProperty method 12.4.1
of JComponent 12.7.5
put*DataType* methods
of Preferences 10.5
putDouble method
of Preferences 10.5
putFloat method
of Preferences 10.5
putInt method
of Preferences 10.5
putLong method
of Preferences 10.5
putNextEntry method 2.2.3
of ZipOutputStream 2.7.7
putValue method 10.4.5
of Action 10.5
put*Xxx* method
of ByteBuffer 2.7.7
put*Xxx* methods
of ByteBuffer 2.4.1

Q

QBE 5.2
QuadCurve2D class 12.5.2, 12.5.3
QuadCurve2D.Double class 12.5.2, 12.7.5
Constructor 12.7.5
QuadCurve2D.Float class 12.5.2
Quadratic curves 12.5.3
quadTo method 12.5.3
of Path2D.Float 12.7.5
Qualified names 3.6
Queries 5.2
by example 5.2
executing 5.4.1, 5.5
multiple 5.4.2
populating row sets with results of 5.7.2
preparing 5.5.1
returning multiple results 5.5.4

quoteReplacement method
of Matcher 2.7.7

R

R 8.2
Race conditions 1.13
Radio buttons 11.4.2, 11.5.3
Random class 9.4.2
methods of 1.12
Random numbers, streams of 1.2, 1.4, 1.12
Random-access files 2.2.2
RandomAccessFile class 2.2.2, 2.4.1, 2.7.7
Constructor 2.7.7
getChannel method 2.7.7
getFilePointer method 2.2.2, 2.7.7
length method 2.2.2, 2.7.7
seek method 2.2.2, 2.7.7
RandomGenerator interface 1.13
doubles method 1.13
equiDoubles method 1.13
ints method 1.13
longs method 1.13
Randomness 9.4.2
range method
of IntStream 1.13
of LongStream 1.13
range, rangeClosed methods
of *Xxx*Stream 1.12
rangeClosed method
of IntStream 1.13
of LongStream 1.13
Ranges, converting to streams 1.13
Raster class 12.7.5
getDataElements method 12.6.4, 12.7.5
getPixel method 12.7.5
getPixel, getPixels methods 12.6.4
getPixels method 12.7.5
Raster images 12.6
constructing from pixels 12.6.4
filtering 12.6.5
readers/writers for 12.6.1
read method
of CipherInputStream 9.4.4
of FileInputStream 2.1.1
of ImageIO 12.6.1, 12.7.5
of ImageReader 12.7.5
of InputStream 2.1.1, 2.7.7
of Readable 2.1.2, 2.7.7
of ReadableByteChannel 2.7.7

of Reader 2.1.2
Readable interface 2.1.2, 2.7.7
read method 2.1.2, 2.7.7
ReadableByteChannel 2.7.7
read method 2.7.7
ReadableByteChannel interface 4.2.4
readAllBytes method 2.1.1
of Files 2.7.7
of InputStream 2.7.7
readAllLines method
of Files 2.7.7
readAll*Xxx* methods
of Files 2.3.2
readAttributes method 2.3.5
of Files 2.7.7
readBoolean method 2.2.1
of DataInput 2.7.7
readByte method
of DataInput 2.7.7
readChar method 2.2.1, 2.2.2
of DataInput 2.7.7
readDouble method 2.2.1, 2.6.1, 2.6.4
of DataInput 2.7.7
Reader class 2.1, 2.1.2
read method 2.1.2
readExternal method 2.6.5
readFixedString method 2.2.2
readFloat method 2.2.1
of DataInput 2.7.7
readFully method
of DataInput 2.7.7
readInt method 2.2.1, 2.2.2, 2.6.1
of DataInput 2.7.7
readLong method 2.2.1
of DataInput 2.7.7
readNBytes method 2.1.1
of InputStream 2.7.7
readObject method
of HashSet 2.6.4
of ObjectInputStream 2.6.1, 2.6.4, 2.7.7
ReadOnlyBufferException class 2.4.1
readResolve method 2.6.6
readShort method 2.2.1
of DataInput 2.7.7
readString method 2.3.2
of Files 2.7.7
readThumbnail method
of ImageReader 12.7.5
readUTF method 2.2.1
of DataInput 2.7.7

REAL 5.2, 5.9.4
Records 2.1.7
deserializing 2.6.8
fixed-size 2.2.2
ignoring serialVersionUID fields 2.6.7
Rectangle class 10.3.1
Rectangle2D class 10.3.1, 12.5.2
Rectangle2D.Double class 10.3.1, 10.5
Constructor 10.5
Rectangle2D.Double, Rectangle2D.Float classes 12.5.2
Rectangle2D.Float class 10.3.1
Rectangles 10.3.1
drawing 10.3.1
filling with color 10.3.2
RectangularShape class 10.3.1, 10.5, 12.5.2
getCenterX method 10.5
getCenterX/Y methods 10.3.1
getCenterY method 10.5
getHeight method 10.5
getHeight/Width methods 10.3.1
getMaxX method 10.5
getMaxY method 10.5
getMinX method 10.5
getMinY method 10.5
getWidth method 10.5
getX method 10.5
getY method 10.5
Redirects, of URLs 4.3.3
reduce method
of Stream 1.10, 1.13
reducing method
of Collectors 1.9.4
Reductions 1.6, 1.10
ref attribute (XML Schema) 3.4.2
Reflection
accessing
protected methods 9.1.2
constructing class trees 12.4.5
enumerating fields from a variable 12.4.6
regexFilter method 12.2.7
of RowFilter 12.7.5
Regular expressions 2.7
escapes in 2.1.7
filtering 12.2.7
finding matches of 2.7.2
flags for 2.7.7
groups in 2.7.4
in DTDs 3.4.1
splitting sequences with 1.2
reinterpret method

of MemorySegment 13.8.3
relative method
of ResultSet 5.6.1, 5.10
Relativization, in URLs 4.3.1
relativize method
of Path 2.3.1, 2.7.7
releaseSavepoint method 5.9.2
of Connection 5.10
ReleaseStringUTFChars 13.1
reload method
of DefaultTreeModel 12.4.2, 12.7.5
remaining method
of Buffer 2.4.2, 2.7.7
remove method
of AttributeSet 12.7.5
of JMenu 11.8.4
removeAllItems method 11.4.4
of JComboBox 11.8.4
removeCellEditorListener method
of CellEditor 12.7.5
removeColumn method 12.2.8
of JTable 12.7.5
removeItem method 11.4.4
of JComboBox 11.8.4
removeItemAt method 11.4.4
of JComboBox 11.8.4
removeLayoutComponent method
of LayoutManager 11.8.4
removeNodeFromParent method 12.4.2
of DefaultTreeModel 12.7.5
removePropertyChangeListener method 10.4.5
removeTreeModelListener method 12.4.6
of TreeModel 12.7.5
RenderableImage 12.7.3
Rendering
cells 12.3.1
columns 12.2.1
headers 12.3.2
nodes 12.4.4
Rendering pipeline (AWT) 12.5.1
Renjin project 8.2, 8.2.1
repaint method
of Component 10.3, 10.5
replaceAll method
of Matcher 2.7.6, 2.7.7
of String 2.7.6
replaceFirst method
of Matcher 2.7.6, 2.7.7
#REQUIRED 3.4.1
required option (JAAS configuration) 9.2.1

requisite option (JAAS configuration) 9.2.1
RescaleOp class 12.6.5, 12.7.5
Constructor 12.7.5
Rescaling operation 12.6.5
reset method
of Buffer 2.4.2, 2.7.7
of InputStream 2.7.7
of MessageDigest 9.4.4
resetChoosableFileFilters method
of JFileChooser 11.8.4
resetChoosableFilters method 11.8.4
resolve method
of Path 2.3.1, 2.7.7
resolveEntity method 3.4.1
of EntityResolver 3.9
resolveSibling method 2.3.1
of Path 2.7.7
Resolving
classes 9.1.1
relative URLs 4.3.1
Resource bundles 7.8
loading 7.8.2, 7.8.3
locating 7.8.1
lookup tables for 7.8.3
naming 7.8.3
searching for 7.8.3
Resource editors 7.8
ResourceBundle class 7.9
extending 7.8.3
getBundle method 7.8.1, 7.9
getKeys method 7.8.3, 7.9
getObject method 7.8.3, 7.9
getString method 7.8.2, 7.9
getStringArray method 7.9
handleGetObject method 7.8.3, 7.9
ResourceBundle.Control class
getCandidateLocales method 7.8.1
Resources
hierarchy of 7.8.1
Response headers 4.3.2
Response pages 4.3.3
Result interface 3.9, 5.9.4
Result sets
accessing columns in 5.4.1
analyzing 5.4.1
closing 5.4.2
for multiple queries 5.4.2
iterating over rows in 5.6
metadata for 5.8
numbering rows in 5.6.1

order of rows in 5.4.1
retrieving multiple 5.5.4
scrollable 5.6.1
updatable 5.6, 5.6.2
results method
of Matcher 2.7.7
ResultSet interface 5.7, 5.10
absolute method 5.6.1, 5.10
afterLast method 5.10
afterLast, beforeFirst methods 5.6.1
beforeFirst method 5.10
cancelRowUpdates method 5.6.2, 5.10
close method 5.4.2, 5.10
concurrency values 5.6.2
deleteRow method 5.6.2, 5.10
findColumn method 5.10
first method 5.6.1, 5.10
getBlob method 5.10
getBlob, getClob methods 5.5.2
getClob method 5.10
getConcurrency method 5.6.1, 5.6.2, 5.10
getDate, getDouble, getInt methods 5.4.1
getMetaData method 5.8, 5.10
getObject method 5.4.1, 5.10
getRow method 5.6.1, 5.10
getString method 5.4.1
getType method 5.6.1, 5.10
get*Xxx* method 5.10
insertRow method 5.6.2, 5.10
isAfterLast method 5.10
isAfterLast, isBeforeFirst methods 5.6.1
isBeforeFirst method 5.10
isClosed method 5.10
isFirst method 5.10
isFirst, isLast methods 5.6.1
isLast method 5.10
iteration protocol 5.4.1
last method 5.6.1, 5.10
moveToCurrentRow method 5.10
moveToInsertRow method 5.10
moveToXxxRow methods 5.6.2
next method 5.4.1, 5.6, 5.10
previous method 5.6.1, 5.10
relative method 5.6.1, 5.10
type values 5.6.1, 5.6.2
updateObject method 5.10
updateRow method 5.10
update*Xxx* method 5.10
update*Xxx* methods 5.6.2
ResultSetMetaData interface 5.8, 5.10

getColumnCount method 5.10
getColumnDisplaySize method 5.10
getColumnLabel method 5.10
getColumnName method 5.10
getColumn*Xxx* methods 5.8
Retirement calculator example 7.9
Return values, missing 1.6
revalidate method
of JComponent 11.3.1, 11.8.4
rewind method
of Buffer 2.4.2, 2.7.7
RFC 1123 6.6
RFC 2396 4.3.1
RFC 2616 4.3.2
RFC 2911 12.7.5
RFC 821 4.6
RFC 822 6.6
RGB 12.5.9, 12.6.4
Rhino 8.2.1, 8.2.6
Rivest, Ronald 9.3.1, 9.3.3, 9.4.4
Role-based authentication 9.2.2
rollback method
of Connection 5.9.1, 5.10
Root component 2.3.1
Root element 3.2
referencing schemas in 3.4.2
Root node 12.4, 12.4.1
handles for 12.4.1
separating children of 12.4.1
rotate method
of Graphics2D 12.5.7, 12.7.5
Rotation 12.5.7
interpolating pixels and 12.6.5
with center point 12.5.7
Round cap 12.5.5
Round join 12.5.5
Rounded rectangles 12.5.3
RoundRectangle2D class 12.5.2, 12.5.3
RoundRectangle2D.Double class 12.5.2, 12.7.5
Constructor 12.7.5
RoundRectangle2D.Float class 12.5.2
Row sets 5.7
cached 5.7.2, 5.8
constructing 5.7.1
modifying 5.7.2
page size of 5.7.2
RowFilter class 12.2.7, 12.7.5
andFilter method 12.2.7, 12.7.5
dateFilter method 12.2.7, 12.7.5
include method 12.2.7, 12.7.5

notFilter method 12.2.7, 12.7.5
numberFilter method 12.2.7, 12.7.5
orFilter method 12.2.7, 12.7.5
regexFilter method 12.2.7, 12.7.5
RowFilter.Entry class 12.2.7, 12.7.5
getIdentifier method 12.7.5
getModel method 12.7.5
getStringValue method 12.7.5
getValue method 12.7.5
getValueCount method 12.7.5
ROWID 5.9.4
RowId interface 5.9.4
Rows (databases) 5.2
deleting/inserting 5.6.2
iterating through 5.6.2
order of, in result set 5.4.1
retrieving 5.9.4
selecting 5.2
updating 5.6.2
Rows (Swing)
filtering 12.2.7
height of 12.2.4
hiding 12.2.7
margins of 12.2.4
position, in a node 12.4.2
resizing 12.2.4
selecting 12.1.1, 12.2.5
sorting 12.1.1, 12.2.6
RowSet interface 5.7, 5.10
execute method 5.10
getCommand method 5.10
getPassword method 5.10
getURL method 5.10
getUsername method 5.10
setCommand method 5.10
setPassword method 5.10
setURL method 5.10
setUsername method 5.10
set*Xxx* methods 5.7.2
RowSetFactory interface 5.10
createCachedRowSet method 5.7.1, 5.7.2, 5.10
createFilteredRowSet method 5.10
createJdbcRowSet method 5.10
createJoinRowSet method 5.10
createWebRowSet method 5.10
RowSetProvider class 5.10
newFactory method 5.7.1, 5.10
RSA algorithm 9.3.3, 9.4.4
RSA Security, Inc. 9.4.1
rt.jar file 9.1.1

Ruby 8.2
run method
of Tool 8.1.1, 8.2.6
Runtime class
exec method 2.6.8

S

S (short), type code 2.6.2
Sample values of pixels 12.6.4
Sandbox 9.3.7
Save points 5.9.2
Savepoint interface 5.10
getSavepointId method 5.10
getSavepointName method 5.10
SAX (Simple API for XML) parser 3.3, 3.7.1
activating namespace processing in 3.7.1
SAXParseException class 3.9
getColumnNumber method 3.9
getLineNumber method 3.9
SAXParser class 3.9
parse method 3.7.1, 3.9
SAXParserFactory class 3.9
isNamespaceAware method 3.9
isValidating method 3.9
newInstance method 3.9
newInstance, newSAXParser methods 3.7.1
newNSInstance method 3.9
newSAXParser method 3.9
setFeature method 3.7.1
setNamespaceAware method 3.9
setValidating method 3.9
SAXResult class 3.9
SAXSource class 3.9
Constructor 3.9
Scalar functions 5.5.3
Scale 12.5.7
scale method
of Graphics2D 12.5.7, 12.7.5
Scaling 12.5.7
Scanner class 1.13, 2.1.5, 2.7.7
constructor 2.3.1, 4.2.4
findAll method 2.7.7
tokens method 1.2, 1.13
useLocale method 7.1.4, 7.2.1
Scheduling applications
computing dates for 6.3
time zones and 6.2, 6.5
schema element (XML Schema) 3.4.2
Scheme 8.2

ScriptContext interface 8.2.2, 8.2.6
getErrorWriter method 8.2.6
getReader method 8.2.6
getWriter method 8.2.6
getXxx/setXxx methods 8.2.3
setErrorWriter method 8.2.6
setReader method 8.2.6
setWriter method 8.2.6
ScriptEngine interface 8.2.6
createBindings method 8.2.6
eval method 8.2.2, 8.2.6
get method 8.2.6
getContext method 8.2.6
put method 8.2.6
ScriptEngineFactory interface 8.2.6
getExtensions method 8.2.6
getMethodCallSyntax method 8.2.4
getMimeTypes method 8.2.6
getNames method 8.2.6
ScriptEngineManager class 8.2.1, 8.2.6
get method 8.2.6
getEngineByExtension method 8.2.6
getEngineByMimeType method 8.2.6
getEngineByName method 8.2.6
getEngineFactories method 8.2.6
getEngineXxx methods 8.2.1
put method 8.2.6
Scripting engines 8.2.1
adding variable bindings to 8.2.2
calling functions in 8.2.4
implementing Java interfaces 8.2.4
invoking 8.2.1
Scripting languages 8.2
Scripts 8.2
accessing classes in 8.2.4
compiling 8.2.5
executing 8.2.2
invoking 8.2.2
redirecting I/O of 8.2.3
using Java method call syntax in 8.2.4
Scroll panes 11.3.4
with tables 12.1.1
with trees 12.4.2
Scrollbars 11.3.5
scrollPathToVisible method 12.4.2
of JTree 12.7.5
SecretKey interface 9.4.2
SecretKeySpec class 9.4.4
Constructor 9.4.4
Secure random generator 9.4.2

SecureRandom class
setSeed method 9.4.2
Security class 9, 9.4.4
bypassing constructors 2.6.8
bytecode verification 9.1.5
class loaders 9.1
code signing 9
different levels of 9.3
digital signatures 9.3
encryption 9.4
user authentication 9.2
Security manager 9
seek method
of RandomAccessFile 2.2.2, 2.7.7
SegmentAllocator interface 13.8.3
allocate method 13.8.3
allocateFrom method 13.8.3
elements method 13.8.3
SELECT 5.2
executing 5.4.1
for LOBs 5.5.2
multiple, in a query 5.5.4
not supported in batch updates 5.9.3
select attribute (XSLT) 3.9
Selection models 12.2.5
Semantic events 10.4.7
send method
of HttpClient 4.6
sendAsync method 4.4.4
of HttpClient 4.6
separator constant of File class 2.1.3
Separators 2.1.3, 2.3.1
sequence element (XML Schema) 3.4.2
sequenceElement method
of MemoryLayout.PathElement 13.8.3
SequenceLayout interface 13.5
sequenceLayout method
of MemoryLayout 13.8.3
Sequences, producing 1.2
Serial numbers 2.6.1
SerialCloneable 2.6.9
Serializable interface 2.6.1, 2.6.2
readResolve, writeReplace methods 2.6.6
Serialization 2.6
cloning with 2.6.9
file format for 2.6.2
filters for 2.6.8
serial numbers for 2.6.1
serialver program 2.6.7
serialVersionUID method 2.6.7

Server-side programs 4.3.3
redirecting URLs in 4.3.3
Servers
accessing 4.3
connecting clients to 4.1.2
echo 4.2.1
implementing 4.2
ServerSocket class 4.2, 4.6
accept method 4.2.1, 4.2.2, 4.6
close method 4.6
Constructor 4.6
ServerSocketFactory class 4.6
createServerSocket method 4.6
Service provider interfaces 12.6.2
Servlets 8.1.6
Session class
setDebug method 4.6
set method
of MemorySegment 13.8.3
setAccelerator method 11.5.5
of JMenuItem 11.8.4
setAcceptAllFileFilterUsed method 11.8.4
of JFileChooser 11.8.4
setAccessory method
of JFileChooser 11.8.4
setAction method
of AbstractButton 11.8.4
setActionCommand method
of AbstractButton 11.8.4
setAllowsChildren method 12.4.1
of DefaultMutableTreeNode 12.7.5
setAllowUserInteraction method 4.3.2
setAsksAllowsChildren method 12.4.1
of DefaultTreeModel 12.7.5
setAtIndex method
of MemorySegment 13.8.3
setAttribute method
of Element 3.9
setAttribute, setAttributeNS methods
of Element 3.8.1
setAttributeNS method
of Element 3.9
setAutoCommit method 5.9.1, 5.9.3
of Connection 5.10
setAutoCreateRowSorter method 12.1.1, 12.2.6
of JTable 12.7.5
setAutoResizeMode method 12.2.3
of JTable 12.7.5
setBackground method 10.3.2
of Component 10.5

setBinaryStream method
of Blob 5.10
setBorder method 11.4.3
of JComponent 11.8.4
setBounds method 10.2.2
of Component 10.5
setCellEditor method 12.3.3
of TableColumn 12.7.5
setCellRenderer method
of TableColumn 12.7.5
setCellSelectionEnabled method 12.2.5
of JTable 12.7.5
setCharacterStream method
of Clob 5.10
setClip method 12.5.8, 12.7.1
of Graphics 12.7.5
setClosedIcon method 12.4.4
of DefaultTreeCellRenderer 12.7.5
setColumns method
of JTextArea 11.3.4, 11.8.4
of JTextField 11.3.1, 11.8.4
setColumnSelectionAllowed method 12.2.5
of JTable 12.7.5
setCommand method 5.7.2
of RowSet 5.10
setComparator method 12.2.6
of DefaultRowSorter 12.7.5
setComponentPopupMenu method 11.5.4
of JComponent 11.8.4
setComposite method 12.5.1, 12.5.9
of Graphics2D 12.7.5
setConnectTimeout method 4.3.2
of URLConnection 4.6
setContentHandler method
of XMLReader 3.9
setContextClassLoader method 9.1.2
of Thread 9.4.4
setCrc method
of ZipEntry 2.7.7
setCurrency method 7.2.3
setCurrentDirectory method 11.8.4
of JFileChooser 11.8.4
setCursor method
of Component 10.5
setDataElements method 12.6.4
of WritableRaster 12.7.5
setDate method 5.5.1
setDebug method 4.6
setDecomposition method
of Collator 7.9

setDefault method
of CookieHandler 4.3.3
of Locale 7.1.3, 7.9
setDefaultButton method 11.8.3
of JRootPane 11.8.4
setDefaultCloseOperation method 11.8.2
setDefaultNamespace method
of XMLStreamWriter 3.9
setDefaultRenderer method 12.3.1
setDisplayedMnemonicIndex method 11.5.5
of AbstractButton 11.8.4
setDoInput method 4.3.2
of URLConnection 4.6
setDoOutput method 4.3.2, 4.3.3
of URLConnection 4.6
setDouble method 5.5.1
setEchoChar method
of JPasswordField 11.8.4
setEditable method 12.4.2
of JComboBox 11.4.4, 11.8.4
of JTextComponent 11.8.4
setEnabled method
of Action 10.4.5, 10.5
of JMenuItem 11.5.6, 11.8.4
setEntityResolver method 3.4.1
of DocumentBuilder 3.9
setErrorHandler method
of DocumentBuilder 3.9
setErrorWriter method
of ScriptContext 8.2.6
setFeature method 3.7.1
setFileFilter method 11.8.4
of JFileChooser 11.8.4
setFileSelectionMode method 11.8.4
of JFileChooser 11.8.4
setFileView method 11.8.4
of JFileChooser 11.8.4
setFillsViewportHeight method
of JTable 12.7.5
setFont method
of JComponent 11.8.4
setForeground method 10.3.2
of Component 10.5
setFrameFromCenter method 10.3.1
setFrom method 4.6
setGroupingUsed method
of NumberFormat 7.9
setHeaderRenderer method
of TableColumn 12.7.5
setHeaderValue method

of TableColumn 12.7.5
setHeaderXxx methods
of TableColumn 12.3.2
setHorizontalTextPosition method 11.5.2
of AbstractButton 11.8.4
setIcon method
of JLabel 11.8.4
of JMenuItem 11.5.2
setIconImage method 10.2.2
of Frame 10.5
setIfModifiedSince method 4.3.2
of URLConnection 4.6
setIgnoringElementContentWhitespace method 3.4.1
of DocumentBuilderFactory 3.9
setInheritsPopupMenu method 11.5.4
of JComponent 11.8.4
setInput method
of ImageReader 12.7.5
setInstanceFollowRedirects method 4.3.3
setInt method 5.5.1
setInverted method 11.4.5
setJMenuBar method 11.5.1
of JFrame 11.8.4
setLabelTable method 11.4.5
of JSlider 11.8.4
setLayout method
of Container 11.8.4
setLeafIcon method 12.4.4
of DefaultTreeCellRenderer 12.7.5
setLevel method
of ZipOutputStream 2.7.7
setLineWrap method 11.3.4
of JTextArea 11.8.4
setLocale method
of MessageFormat 7.9
setLocation method 10.2.2
of Component 10.5
setLogWriter method 5.3.4
setMajorTickSpacing method
of JSlider 11.8.4
setMaximumFractionDigits method
of NumberFormat 7.9
setMaximumIntegerDigits method
of NumberFormat 7.9
setMaxWidth method 12.2.3
of TableColumn 12.7.5
setMethod method
of ZipEntry 2.7.7
of ZipOutputStream 2.7.7
setMinimumFractionDigits method

of NumberFormat 7.9
setMinimumIntegerDigits method
of NumberFormat 7.9
setMinorTickSpacing method
of JSlider 11.8.4
setMinWidth method 12.2.3
of TableColumn 12.7.5
setMnemonic method 11.5.5
of AbstractButton 11.8.4
setModel method 11.4.4
setMultiSelectionEnabled method 11.8.4
of JFileChooser 11.8.4
setName method
of NameCallback 9.4.4
setNamespaceAware method
of DocumentBuilderFactory 3.4.2, 3.6, 3.7.1, 3.8.2, 3.9
of SAXParserFactory 3.9
setOpenIcon method 12.4.4
of DefaultTreeCellRenderer 12.7.5
setOutput method
of ImageWriter 12.7.5
setOutputProperty method 3.8.3
of Transformer 3.9
setPageable method 12.7.2
setPageSize method 5.7.2
of CachedRowSet 5.10
setPaint method 10.3.2, 12.5.1, 12.5.6
of Graphics2D 10.5, 12.7.5
setPaintLabels method 11.4.5
of JSlider 11.8.4
setPaintTicks method 11.4.5
of JSlider 11.8.4
setPaintTrack method
of JSlider 11.8.4
setParseIntegerOnly method
of NumberFormat 7.9
setPassword method
of PasswordCallback 9.4.4
of RowSet 5.7.2, 5.10
setPixel method
of WritableRaster 12.7.5
setPixel, setPixels methods
of WritableRaster 12.6.4
setPixels method
of WritableRaster 12.7.5
setPreferredWidth method 12.2.3
of TableColumn 12.7.5
setPrefix method
of XMLStreamWriter 3.9
setPrintable method

of PrinterJob 12.7.5
setProperty method 3.7.2
of XMLInputFactory 3.9
setReader method 8.2.3
of ScriptContext 8.2.6
setReadTimeout method 4.3.2
of URLConnection 4.6
setRenderingHint, setRenderingHints methods
of Graphics2D 12.5.1
setRequestProperty method 4.3.2
of URLConnection 4.6
setResizable method 10.2.2, 12.2.3
of Frame 10.5
of TableColumn 12.7.5
setRootVisible method 12.4.1
of JTree 12.7.5
setRowFilter method 12.2.7
of DefaultRowSorter 12.7.5
setRowHeight method
of JTable 12.7.5
setRowHeight, setRowMargin methods
of JTable 12.2.4
setRowMargin method
of JTable 12.7.5
setRows method 11.3.4
of JTextArea 11.8.4
setRowSelectionAllowed method 12.2.5
of JTable 12.7.5
setRowSorter method 12.2.6
of JTable 12.7.5
setSavepoint method
of Connection 5.10
setSeed method 9.4.2
setSelected method
of AbstractButton 11.8.4
of JCheckBox 11.4.1, 11.8.4
setSelectedFile method
of JFileChooser 11.8.4
setSelectedFile/Files methods
of JFileChooser 11.8.4
setSelectedFiles method
of JFileChooser 11.8.4
setSelectionMode method 12.2.5
of ListSelectionModel 12.7.5
setShowsRootHandles method 12.4.1
of JTree 12.7.5
setSize method
of Component 10.5
of ZipEntry 2.7.7
setSnapToTicks method 11.4.5

of JSlider 11.8.4
setSortable method 12.2.6
of DefaultRowSorter 12.7.5
setSoTimeout method 4.1.3
of Socket 4.6
setStrength method
of Collator 7.9
setString method
of PreparedStatement 5.5.1
of RowSet 5.7.2
setStringConverter method
of TableRowSorter 12.7.5
setStroke method 12.5.1, 12.5.5
of Graphics2D 12.7.5
setSubject method 4.6
setTableName method 5.7.2
of CachedRowSet 5.10
setTabSize method
of JTextArea 11.8.4
setText method 4.6
of JLabel 11.8.4
of JTextComponent 11.3.1, 11.8.4
setTitle method 10.2.2
of Frame 10.5
setToolTipText method
of JComponent 11.8.4
setToRotation method
of AffineTransform 12.7.5
setToScale method
of AffineTransform 12.7.5
setToShear method
of AffineTransform 12.7.5
setToTranslation method
of AffineTransform 12.7.5
setToXxx methods
of AffineTransform 12.5.7
setTransform method 12.5.7
of Graphics2D 12.7.5
setURL method 5.7.2
of RowSet 5.10
setUseCaches method 4.3.2
setUsername method 5.7.2
of RowSet 5.10
setUserObject method 12.4.1
of MutableTreeNode 12.7.5
setValidating method
of DocumentBuilderFactory 3.4.1, 3.9
of SAXParserFactory 3.9
setValueAt method 12.3.4
of TableModel 12.7.5

setVisible method
of Component 10.2.1, 10.5
of JDialog 11.8.2, 11.8.3
setWidth method 12.2.3
of TableColumn 12.7.5
setWrapStyleWord method
of JTextArea 11.8.4
setWriter method 8.2.3
of ScriptContext 8.2.6
set*Xxx* method
of PreparedStatement 5.10
SGML 3.1
SHA-1 2.6.2, 9.3.1
Shamir, Adi 9.3.3, 9.4.4
Shape interface 10.3.1, 12.5.2
implementing 12.5.4
ShapeMaker
getPointCount method 12.5.3
makeShape method 12.5.3
ShapePanel 12.5.3
Shapes
clipping 12.5.1, 12.5.8
combining 12.5.1, 12.5.4
control points of 12.5.3
drawing 12.5.1, 12.5.2
filling 12.5.1, 12.5.6
rendering 12.5.1
transforming 12.5.1
Shared arena 13.3
Shared libraries 13.1
Shear 12.5.7
shear method
of Graphics2D 12.5.7, 12.7.5
Shift-JIS 2.1.8
short type
printing 2.1.6
streams of 1.12
type code for 2.6.2
writing in binary format 2.2.1
ShortBuffer class 2.4.2
ShortLookupTable class 12.6.5, 12.7.5
Constructor 12.7.5
shouldSelectCell method 12.3.4
of CellEditor 12.7.5
show method
of JPopupMenu 11.5.4, 11.8.4
showConfirmDialog method 11.8.1
of JOptionPane 11.8.4
showDialog method 11.8.3, 11.8.4
of JFileChooser 11.8.4

showInputDialog method 11.8.1
of JOptionPane 11.8.4
showInternalConfirmDialog method
of JOptionPane 11.8.4
showInternalInputDialog method
of JOptionPane 11.8.4
showInternalMessageDialog method
of JOptionPane 11.8.4
showInternalOptionDialog method
of JOptionPane 11.8.4
showMessageDialog method 11.8.1
of JOptionPane 11.8.4
showOpenDialog method 11.8.4
of JFileChooser 11.8.4
showOptionDialog method 11.8.1
of JOptionPane 11.8.4
showSaveDialog method 11.8.4
of JFileChooser 11.8.4
shutdownInput method
of Socket 4.6
shutdownOutput method
of Socket 4.6
Signatures. See Digital signatures
Simple types 3.4.2
SimpleDateFormat class 7.5.1
SimpleDoc class 12.7.3, 12.7.5
Constructor 12.7.5
SimpleFileVisitor class 2.3.8, 2.7.7
postVisitDirectory method 2.7.7
preVisitDirectory method 2.7.7
visitFile method 2.3.8, 2.7.7
visitFileFailed method 2.3.8, 2.7.7
SimpleJavaFileObject class 8.1.5, 8.2.6
getCharContent method 8.2.6
openOutputStream method 8.2.6
SimpleScriptContext class 8.2.2
simpleType method 3.4.2
Singletons 2.6.6
size method
of BasicFileAttributes 2.7.7
of Files 2.3.5, 2.7.7
skip method
of InputStream 2.7.7
of Stream 1.4, 1.13
skipBytes method
of DataInput 2.7.7
skipNBytes method
of InputStream 2.7.7
Sliders 11.4.5
ticks on 11.4.5

vertical 11.4.5
SMALLINT 5.2, 5.9.4
SMTP 4.6
Socket class 4.6
connect method 4.6
Constructor 4.6
getInputStream method 4.1.2, 4.2.1, 4.6
getOutputStream method 4.2.1, 4.6
isClosed method 4.6
isConnected method 4.6
isInputShutdown method 4.6
isOutputShutdown method 4.6
setSoTimeout method 4.1.3, 4.6
shutdownInput method 4.6
shutdownOutput method 4.6
SocketChannel class 4.2.4, 4.6
open method 4.2.4, 4.6
SocketFactory class 4.6
createSocket method 4.6
Sockets
half-closing 4.2.3
interrupting 4.2.4
opening 4.1.2
server 4.2.1
timeouts of 4.1.3
SocketTimeoutException class 4.1.3
SoftBevelBorder class 11.4.3, 11.8.4
Constructor 11.8.4
Software developer certificates 9.3.7
sort method
of Collections 7.4
sorted method
of Stream 1.5, 1.13
Source files
character encoding of 7.7.5
reading from memory 8.1.4
Source interface 3.9, 5.9.4
Space. See Whitespace
SPARC processor, big-endian order in 2.2.1
split method
of Pattern 2.7.5, 2.7.7
of String 1.13, 2.1.7, 2.7.5
splitAsStream method 1.2, 2.7.5
of Pattern 1.13, 2.7.7
spliterator method
of Iterable 1.13
Spliterators class 1.13
spliteratorUnknownSize method 1.2, 1.13
spliteratorUnknownSize method
of Spliterators 1.13

splitWithDelimiters method
of Pattern 2.7.7
Spring layout 11.6
SQL 5, 5.2
changing data inside databases 5.2
commands in 5.3.3
data types in 5.2, 5.9.4
equality testing in 5.2
escapes in 5.5.3
exceptions in 5.4.3
executing statements in 5.4.1
keywords in 5.2
reading instructions from a file 5.4.4
strings in 5.2
vs. Java 5.5.1
warnings in 5.4.3
wildcards in 5.2
SQLException class 5.4.3, 5.6.1, 5.10
getErrorCode method 5.10
getNextException method 5.10
getSQLState method 5.10
get*Xxx* methods 5.4.3
iterator method 5.4.3, 5.10
rolling back and 5.9.1
SQLWarning class 5.4.3, 5.6.1, 5.10
getNextWarning method 5.10
SQLXML interface 5.9.4
Square cap 12.5.5
Square root, computing 1.7.6
SQuirreL 5.8
SRC, SRC_Xxx composition rules 12.5.9
SRGB standard 12.6.4
SSLServerSocketFactory class 4.6
getDefault method 4.6
SSLSocketFactory class 4.6
getDefault method 4.6
Standard extensions 9.1.1
Standard input 2.1.1
StandardCharsets class 2.1.8
StandardJavaFileManager interface 8.1.2, 8.1.3, 8.1.5, 8.2.6
getJavaFileObjectsFromFiles method 8.2.6
getJavaFileObjectsFromPaths method 8.2.6
getJavaFileObjectsFromStrings method 8.2.6
start method
of MatchResult 2.7.7
startDocument method 3.7.1
of ContentHandler 3.9
startElement method 3.7.1
of ContentHandler 3.9
stateChanged method 11.4.5

Stateless operations 1.13
Statement class 5.4.1
addBatch method 5.9.3
close, closeOnCompletion methods 5.4.2
execute method 5.4.4, 5.5.4
executeBatch method 5.9.3
executeQuery method 5.4.1, 5.6.1, 5.6.2
executeUpdate method 5.4.1, 5.9.1, 5.9.2
getMoreResults method 5.5.4
getUpdateCount method 5.5.4
using for multiple queries 5.4.2
Statement interface 5.10
addBatch method 5.10
close method 5.10
closeOnCompletion method 5.10
execute method 5.10
executeBatch method 5.10
executeLargeBatch method 5.10
executeLargeUpdate method 5.10
executeQuery method 5.10
executeUpdate method 5.10
getLargeUpdateCount method 5.10
getMoreResults method 5.10
getResultSet method 5.10
getUpdateCount method 5.10
isClosed method 5.10
Statements
closing 5.4.2
complex 5.5.1
concurrently open 5.4.2
executing 5.4.1
grouping into transactions 5.9
in batch updates 5.9.3
multiple 5.4.2
prepared 5.5.1
truncations in 5.4.3
Static initialization blocks 13.1
Static inner classes 2.6.2
statusCode method 4.4.3
of HttpResponse 4.6
StAX parser 3.7.2, 3.8.4
namespace processing in 3.7.2
no indented output in 3.8.4
StAXSource 3.9
stopCellEditing method 12.3.4
of CellEditor 12.7.5
Stored procedures 5.5.3
Stream 1.13
collect method 1.13
gather method 1.13

reduce method 1.13
Stream interface 1.13
allMatch method 1.13
allMatch, anyMatch methods 1.6
anyMatch method 1.13
collect method 1.9.1, 1.10, 1.13
concat method 1.4, 1.13
count method 1.1, 1.6, 1.13
distinct method 1.5, 1.13
dropWhile method 1.4, 1.13
empty method 1.2, 1.13
filter method 1.1, 1.3, 1.6, 1.13
findAny method 1.6, 1.13
findFirst method 1.6, 1.13
flatMap method 1.3, 1.13
forEach method 1.8, 1.13
forEachOrdered method 1.8
generate method 1.2, 1.12, 1.13
iterate method 1.2, 1.5, 1.12, 1.13
iterator method 1.8
limit method 1.4, 1.13
map method 1.3, 1.13
mapMulti method 1.13
mapToInt method 1.10
max method 1.13
max, min methods 1.6
min method 1.13
noneMatch method 1.6, 1.13
of method 1.2, 1.13
ofNullable method 1.2, 1.7.7, 1.13
peek method 1.5, 1.13
reduce method 1.10
skip method 1.4, 1.13
sorted method 1.5, 1.13
takeWhile method 1.4, 1.13
toArray method 1.8, 1.13
toList method 1.2, 1.8, 1.13
unordered method 1.13
stream method
of Arrays 1.2, 1.12, 1.13
of Collection 1.1, 1.13
of Optional 1.7.7
of StreamSupport 1.2, 1.13
Streaming parsers 3.3, 3.7
StreamPrintService class 12.7.4
StreamPrintServiceFactory class 12.7.4, 12.7.5
getPrintService method 12.7.4, 12.7.5
lookupStreamPrintServiceFactories method 12.7.5
StreamResult class 3.9
Constructor 3.9

Streams class 1
collecting results from 1.8
computing values from 1.10
converting to/from arrays 1.2, 1.8, 1.13
creating 1.2
debugging 1.5
empty 1.2, 1.6, 1.10
encrypted 9.4.3
filtering 1.7.7
finite 1.2
flattening 1.3, 1.7.6
for print services 12.7.4
infinite 1.1, 1.2, 1.4, 1.5
intermediate operations for 1.1
noninterference of 1.2
of primitive types 1.10
of random numbers 1.12
parallel 1.1, 1.6, 1.8, 1.9.2, 1.9.3, 1.10, 1.13
processed lazily 1.1, 1.2, 1.5
reductions of 1.6
removing duplicates from 1.5
returned by Files.lines 1.13
sorted 1.5, 1.13
splitting/combining 1.4
summarizing 1.9
terminal operations for 1.1, 1.6
transformations of 1.3, 1.12
vs. collections 1.1
StreamSource class 3.9
Constructor 3.9
transform method 3.9
StreamSupport class 1.13
stream method 1.2, 1.13
String class 1.13, 2.1.2
compareTo method 7.4
format method 7.1.4
lines method 1.13
replaceAll method 2.7.6
split method 1.13, 2.1.7, 2.7.5
strip method 3.3, 7.2.1, 11.3.1
toLowerCase method 1.3, 7.1.4
toUpperCase method 7.1.4
String parameters 13.1
StringBuffer class 2.1.2, 2.4.2
StringBuilder class 2.1.2, 2.2.2
Strings
converting to grapheme clusters 1.3
encoding 7, 7.7.5
fixed-size, I/O of 2.2.2
in native code 13.1

in SQL 5.2
internationalizing 7.8
ordering 7.4
patterns for 2.7
printing 2.1.6
sorting 7.4
splitting 1.2
transforming to lower/uppercase 1.3, 7.1.4
writing in binary format 2.2.1
StringSource 8.1.4
strip method
of String 3.3, 7.2.1, 11.3.1
Stroke interface 12.5.5
StrokePanel 12.5.5
Strokes 12.5.1, 12.5.5
dashed 12.5.5
setting 12.5.1
styling 12.5.5
StructLayout interface 13.5
structLayout method
of MemoryLayout 13.8.3
Stylesheets 3.9
Subject class 9.4.4
getPrincipals method 9.4.4
Subjects (logins) 9.2.1
Submenus 11.5, 11.5.1
subSequence method
of CharSequence 2.7.7
subtract method
of Area 12.5.4, 12.7.5
Subtraction operator, not associative 1.10
Subtrees 12.4.1, 12.4.4
adding nodes to 12.4.2
collapsed and expanded 12.4.1
Suetonius, Gaius Tranquillus 9.1.4
sufficient option (JAAS configuration) 9.2.1
sum method
of DoubleStream 1.13
of IntStream 1.13
of LongStream 1.13
sum, summaryStatistics methods (primitive streams) 1.12
summarizingDouble method
of Collectors 1.13
summarizingInt method
of Collectors 1.13
summarizingLong method
of Collectors 1.13
summarizing*Xxx* methods
of Collectors 1.9, 1.9.4
summaryStatistics method

of DoubleStream 1.13
of IntStream 1.13
of LongStream 1.13
summingDouble method
of Collectors 1.13
summingInt method
of Collectors 1.13
summingLong method
of Collectors 1.13
summing*Xxx* methods
of Collectors 1.9.4
Sun 5
Sun Microsystems 10.1
SunJCE ciphers 9.4.1
Superclasses
adding 2.6.7
not serializable 2.6.1
Supplier interface 1.13
get method 1.13
supplier method
of Collector 1.13
SupportedValuesAttribute interface 12.7.5
supportsBatchUpdates method 5.9.3
of DatabaseMetaData 5.10
supportsResultSetConcurrency method
of DatabaseMetaData 5.10
supportsResultSetType method
of DatabaseMetaData 5.10
supportsResultSet*Xxx* methods (DatabaseMetaData) 5.6.1
SVG 3.8.4
Swing 10, 12
building GUI with 11
generating dynamic code for 8.1.6
model-view-controller analysis of 11.1
starting 10.2.1
tables in 12.1
trees in 12.4
SwingConstants interface 11.3.2
SwingUtilities class 11.8.4
getAncestorOfClass method 11.8.4
SwingUtilities class, getAncestorOfClass method 11.8.3
SymbolLookup interface 13.6, 13.8.3
find method 13.8.3
findOrThrow method 13.8.3
libraryLookup method 13.8.3
Symmetric ciphers 9.4.1
performance of 9.4.4
SyncProviderException 5.7.2
SYSTEM 3.4.1, 3.8.3
System class 13.8.3

console method 7.7.3
loadLibrary method 13.1, 13.8.3
System class loader 9.1.1
System.err 2.1.6, 8.1.1
System.in 2.1.1, 2.1.6
character encoding and 7.7.3
System.out 2.1.6, 8.1.1
character encoding and 7.7.3
println method 7.7.2, 7.7.3
systemNodeForPackage method
of Preferences 10.5
systemNodeForPackage, systemRoot methods of Preferences
of Preferences 10.5
systemRoot method
of Preferences 10.5

T

t (SQL escape) 5.5.3
Table cell renderers 12.2.1
Table models 12.1.1, 12.1.2
updating after cell editing 12.3.4
TableCellEditor interface 12.7.5
getTableCellEditorComponent method 12.3.4, 12.7.5
implementing 12.3.4
TableCellRenderer interface 12.7.5
getTableCellRendererComponent method 12.3.1, 12.7.5
implementing 12.3.1
TableColumn class 12.2.2, 12.2.3, 12.2.8, 12.7.5
Constructor 12.7.5
setCellEditor method 12.3.3, 12.7.5
setCellRenderer method 12.7.5
setHeaderRenderer method 12.7.5
setHeaderValue method 12.7.5
setHeaderXxx methods 12.3.2
setMaxWidth method 12.7.5
setMinWidth method 12.7.5
setPreferredWidth method 12.7.5
setResizable method 12.7.5
setResizable, setXxxWidth methods 12.2.3
setWidth method 12.2.3, 12.7.5
TableColumnModel interface 12.2.2, 12.7.5
getColumn method 12.2.2, 12.7.5
TableModel interface 12.2.6, 12.7.5
getColumnClass method 12.2.1, 12.7.5
getColumnCount method 12.7.5
getColumnName method 12.7.5
getColumnType method 12.3.3
getRowCount method 12.7.5
getValueAt method 12.1.2, 12.3.3, 12.7.5

getXxxCount methods 12.1.2
implementing 12.1.2
isCellEditable method 12.3.3, 12.7.5
setValueAt method 12.3.4, 12.7.5
TableRowSorter class 12.2.6, 12.2.7, 12.7.5
setStringConverter method 12.7.5
Tables (databases) 5.2
changing data in 5.2
creating 5.2
duplicating data in 5.2
inspecting 5.2
metadata for 5.8
multiple, selecting data from 5.2
removing 5.3.4
Tables (Swing) 12.1
asymmetric 12.2
cells in
editing 12.3.3
rendering 12.3.1
selecting 12.2.5
columns in
accessing 12.2.2
adding 12.2.8
hiding 12.2.8
naming 12.1.2
rearranging 12.1.1
rendering 12.2.1
resizing 12.1.1, 12.2.3
selecting 12.2.5
combo boxes in 12.3.3
constructing 12.1.1, 12.1.2
custom editors in 12.3.4
headers in 12.1.1
rendering 12.3.2
index values in 12.2.5
printing 12.1.1
relationship between classes of 12.2.2
rows in
filtering 12.2.7
hiding 12.2.7
margins of 12.2.4
resizing 12.2.4
selecting 12.1.1, 12.2.5
sorting 12.1.1, 12.2.6
scrolling 12.1.1
text fields in 12.3.3
TableStringConverter class 12.7.5
toString method 12.2.6, 12.7.5
takeWhile method 1.4
of Stream 1.13

TCP 4.1.2
teeing method
of Stream 1.9.4
Telnet 4.1.1
activating/connecting 4.1.1
windows communicating in 4.2.2
template element (XSLT) 3.9
TemporalAdjuster interface 6.3
TemporalAdjusters class 6.3, 6.7
dayOfWeekInMonth method 6.7
firstDayOfMonth method 6.7
firstDayOfNextMonth method 6.7
firstDayOfNextYear method 6.7
firstDayOfYear method 6.7
lastDayOfMonth method 6.7
lastDayOfYear method 6.7
lastInMonth method 6.7
next method 6.7
nextOrSame method 6.3, 6.7
ofDateAdjuster method 6.3
previous method 6.7
previousOrSame method 6.7
Text areas 11.3.4
formatted text in 11.3.5
preferred size of 11.3.4
scrollbars in 11.3.5
Text fields 11.3, 12.3.3
columns in 11.3.1
creating blank 11.3.1
preferred size of 11.3.1
Text files, encoding of 7.7.1, 7.7.4
Text input 11.3
labels for 11.3.2
password fields 11.3.3
scroll panes 11.3.4
Text interface 2.1.4
centering 10.3.3
displaying 10.3
fonts for 10.3.3
generating from XML files 3.9
output 2.1.6
printing 12.7.1, 12.7.3
reading 2.1.5
saving objects in 2.1.7
transmitting through sockets 4.2
typesetting properties of 10.3.3
vs. binary data 2.1.4
Text nodes
constructing 3.8.1
retrieving from XML 3.3

TextCallbackHandler class 9.2.2
TextLayout class 12.5.8, 12.7.5
Constructor 12.7.5
getAdvance method 12.7.5
getAscent method 12.7.5
getDescent method 12.7.5
getLeading method 12.7.5
TextStyle enumeration 7.3
TextSyntax class 12.7.5
TexturePaint class 12.5.6, 12.7.5
Constructor 12.7.5
Thread class 9.4.4
get/setContextClassLoader methods 9.1.2
getContextClassLoader method 9.4.4
setContextClassLoader method 9.4.4
Threads
blocking 2.1.1, 4.2.4
executing scripts in 8.2.2
Internet connections with 4.2.2
race conditions in 1.13
referencing class loaders in 9.1.2
Thumbnails 12.6.3
Ticks 11.4.5
icons for 11.4.5
labeling 11.4.5
snapping to 11.4.5
TIME 5.2, 5.5.3, 5.9.4
current 6.1
formatting 6.6, 7.3
instances of 6.4
literals for 5.5.3
local 6.4
measuring 6.1
parsing 6.6
zoned 6.5, 7.3
Time class 6.7
valueOf method 6.7
Time of day service 4.1.1
Timeouts 4.1.3
Timer class 10.4.7
TIMESTAMP 5.2, 5.5.3, 5.9.4
Timestamp class 6.7
toInstant method 6.7
valueOf method 6.7
Timestamps 6.6
using instants as 6.1
TimeZone class
getTimeZone method 6.7
toZoneId method 6.7
toAbsolutePath method 2.3.1

of Path 2.7.7
toArray method
of AttributeSet 12.7.5
of DoubleStream 1.13
of IntStream 1.13
of LongStream 1.13
of MemorySegment 13.8.3
of primitive streams 1.12
of Stream 1.8, 1.13
toCollection method 1.9.1
of Collectors 1.13
toConcurrentMap method 1.9.2
of Collectors 1.13
toDays method 6.1
toDaysPart method
of Duration 6.7
toFile method 2.3.1
of Path 2.7.7
toFormat method 6.6, 6.7
toHours method 6.1
toInstant method
of Date 6.7
of FileTime 2.7.7, 6.7
of Timestamp 6.7
of ZonedDateTime 6.5, 6.7
tokens method
of Scanner 1.2, 1.13
toLanguageTag method 7.1.2
of Locale 7.9
toList method
of Collectors 1.9.1, 1.13
of Stream 1.2, 1.8, 1.13
toLocalDate method
of ZonedDateTime 6.7
toLocalDateTime method
of ZonedDateTime 6.7
toLocalTime method
of ZonedDateTime 6.7
toLocalXxx methods
of LocalXxx 6.7
toLowerCase method 1.3, 7.1.4
toMap method 1.9.2
of Collectors 1.13
toMillis, toMinutes, toNanos methods
of Duration 6.1
toNanoOfDay method
of LocalTime 6.7
Tool interface 8.2.6
run method 8.1.1, 8.2.6
Toolbars 11.5.7

detaching 11.5.7
dragging 11.5.7
title of 11.5.7
vertical 11.5.7
Toolkit class 10.5
getDefaultToolkit method 10.2.2, 10.5
getScreenSize method 10.2.2, 10.5
ToolProvider class
getSystemJavaCompiler method 8.1.1
Tooltips 11.5.8
toPath method 2.3.1
of File 2.7.7
toSecondOfDay method
of LocalTime 6.7
toSeconds method 6.1
of Duration 6.7
toSet method 1.9.1, 1.9.4
of Collectors 1.13
toString method
of CharSequence 2.7.7
of Currency 7.9
of Locale 7.9
of TableStringConverter 12.2.6, 12.7.5
of Variable 12.4.6
toUnmodifiableList method
of Collectors 1.13
toUnmodifiableMap method
of Collectors 1.13
toUnmodifiableSet method
of Collectors 1.13
toUpperCase method 7.1.4
to*Xxx* method
of Duration 6.7
to*Xxx*Part method
of Duration 6.7
toZonedDateTime method 6.7
toZoneId method 6.7
Transactions 5.9
committing 5.9
error handling in 5.9.3
rolling back 5.9
transferTo method
of InputStream 2.7.7
transform method
of Graphics2D 12.5.1, 12.5.7, 12.7.5
of StreamSource 3.9
of Transformer 3.8.3, 3.9
Transformations 12.5.1, 12.5.7
affine 12.5.7, 12.6.5
composing 12.5.7

fundamental types of 12.5.7
matrices for 12.5.7
order of 12.5.7
setting 12.5.1
using for printing 12.7.2
Transformer class 3.9
setOutputProperty method 3.8.3, 3.9
transform method 3.8.3, 3.9
TransformerFactory class 3.9
newInstance method 3.8.3, 3.9
newTransformer method 3.8.3, 3.9
transient keyword 2.6.3, 2.6.4
Translate 12.7.2
translate method
of Graphics2D 12.5.7, 12.7.5
translatePoint method 10.4.7
Translation 12.5.7
Transparency interface 12.5.9
Traversal order 12.4.3
Tree events 12.4.5
Tree models
constructing 12.4.1, 12.4.6
custom 12.4.6
default 12.4.1
Tree parsers 3.3
Tree paths 12.4.2
constructing 12.4.2, 12.4.3
Tree selection listeners 12.4.5
TreeCellRenderer interface 12.4.4, 12.7.5
getTreeCellRendererComponent method 12.4.4, 12.7.5
implementing 12.4.4
TreeMap class 1.9.2
TreeModel interface 12.4.1, 12.4.2, 12.7.5
add/removeTreeModelListener methods 12.4.6
addTreeModelListener method 12.7.5
getChild method 12.7.5
getChild, getChildCount methods 12.4.6
getChildCount method 12.7.5
getIndexOfChild method 12.4.6, 12.7.5
getRoot method 12.4.6, 12.7.5
implementing 12.4.1
isLeaf method 12.4.6, 12.7.5
removeTreeModelListener method 12.7.5
valueForPathChanged method 12.4.6, 12.7.5
TreeModelEvent class 12.7.5
Constructor 12.7.5
TreeModelListener interface 12.4.6, 12.7.5
treeNodesChanged method 12.7.5
treeNodesInserted method 12.7.5
treeNodesRemoved method 12.7.5

treeNodesXxx methods 12.4.6
treeStructureChanged method 12.4.6, 12.7.5
TreeNode interface 12.4.1, 12.4.2, 12.7.5
children method 12.7.5
getAllowsChildren method 12.7.5
getChildAt method 12.7.5
getChildCount method 12.7.5
getParent method 12.4.3, 12.7.5
isLeaf method 12.4.1, 12.7.5
treeNodesChanged method
of TreeModelListener 12.7.5
treeNodesInserted method
of TreeModelListener 12.7.5
treeNodesRemoved method
of TreeModelListener 12.7.5
TreePath class 12.4.2, 12.7.5
getLastPathComponent method 12.4.2, 12.7.5
Trees 12.4
adding listeners to 12.4.5
background color for 12.4.4
connecting lines in 12.4.1
displaying 12.4.1
editing 12.4.2, 12.4.6
handles in 12.4.1, 12.4.4
hierarchy of classes for 12.4.1
indexes in 12.4.2
infinite 12.4.6
leaves in 12.4, 12.4.1, 12.4.4, 12.4.6
nodes in 12.4, 12.4.1, 12.4.4, 12.4.6
paired with other components 12.4.5
rendering 12.4.4
scrolling to newly added nodes 12.4.2
structure of 12.4
subtrees in 12.4.1
traversals for 12.4.3
updating vs. reloading 12.4.2
user objects for 12.4.1, 12.4.2
view of 12.4.2
with horizontal lines 12.4.1
TreeSelectionEvent class 12.7.5
getPath method 12.7.5
getPaths method 12.4.5, 12.7.5
TreeSelectionListener interface 12.7.5
implementing 12.4.5
valueChanged method 12.4.5, 12.7.5
TreeSelectionModel interface 12.4.5
treeStructureChanged method
of TreeModelListener 12.7.5
Troubleshooting. See Debugging
True Odds: How Risks Affect Your Everyday Life (Walsh) 9.3.1

truncate method
of FileChannel 2.7.7
try-with-resources statement 2.1.2
closing files with 2.3.6, 2.3.8
for database connections 5.4.2
with locks 2.5
tryLock method 2.5
of FileChannel 2.7.7
ts (SQL escape) 5.5.3
Type codes 2.6.2
Type definitions 3.4.2
anonymous 3.4.2
nesting 3.4.2
type method
of XPathEvaluationResult 3.9
TYPE_BICUBIC, TYPE_BILINEAR fields
of AffineTransformOp 12.6.5
TYPE_BYTE_GRAY 12.6.4
TYPE_INT_ARGB 12.6.4
TYPE_NEAREST_NEIGHBOR field
of AffineTransformOp 12.6.5
Types. See Data types
Typesetting terms 10.3.3

U

U.S. government on exporting encryption methods 9.1.4
UDP 4.1.2
UIManager class 12.3.1
UncheckedIOException class 2.1.5
Unicode 1.12, 2.1.8
character order in 7.4
converting to binary data 2.1.4
in property files 7.8.2
input/output streams and 2.1
native code and 13.1
normalization forms in 7.4
using for all strings 7
unionLayout method
of MemoryLayout 13.8.3
Units of measurement 3.2
UNIX
authentication in 9.2.1
line feed in 2.1.6, 7.7.2
paths in 2.3.1
specifying locales in 7.1.3
UnixLoginModule 9.2.1
UnixNumericGroupPrincipal class 9.2.1
UnixPrincipal class 9.2.1
UnknownHostException class 4.1.2

unordered method
of BaseStream 1.13
of Stream 1.13
Unparsed external entities 3.4.1
unread method
of PushbackInputStream 2.7.7
UnsatisfiedLinkError class 13.1
until method
of LocalDate 6.2, 6.7
upcallStub method
of Linker 13.8.3
UPDATE 5.2, 5.5.1, 5.6.2
executing 5.4.1
in batch updates 5.9.3
truncations in 5.4.3
vs. methods of ResultSet 5.6.2
update method
of Cipher 9.4.1, 9.4.2, 9.4.3, 9.4.4
of MessageDigest 9.3.1, 9.4.4
updateObject method
of ResultSet 5.10
updateRow method
of ResultSet 5.10
update*Xxx* method
of ResultSet 5.10
update*Xxx* methods (ResultSet) 5.6.2
URI class 4.4.2
get*Xxx* methods 4.3.1
no resource accessing with 4.3.1
uri method
of HttpRequest.Builder 4.4.2, 4.4.4, 4.6
URIs 3.6, 4.3.1
absolute vs. relative 4.3.1
base 4.3.1
hierarchical 4.3.1
namespace 3.6
opaque vs. nonopaque 4.3.1
schemes for 4.3.1
with HTTP 4.4.2
URISyntax class 12.7.5
URL class 4.3.1, 4.4.2, 4.6
accepted schemes for 4.3.1
openConnection method 4.3.2, 4.6
openStream method 2.1.3, 4.3.1, 4.6
URLClassLoader class 9.4.4
addURLs method 9.1.2
Constructor 9.4.4
getURLs method 9.1.2
loadClass method 9.1.2
URLConnection class 4.3.1, 4.3.2, 4.4, 4.6

connect method 4.3.2, 4.6
getConnectTimeout method 4.6
getContent method 4.6
getContentEncoding method 4.6
getContentEncoding, getContentType methods 4.3.2, 4.3.3
getContentLength method 4.3.2, 4.6
getContentType method 4.6
getDate method 4.3.2, 4.6
getDoInput method 4.6
getDoOutput method 4.6
getExpiration method 4.3.2, 4.6
getHeaderField method 4.6
getHeaderFieldKey method 4.6
getHeaderFields method 4.6
getHeader*Xxx* methods 4.3.2
getIfModifiedSince method 4.6
getInputStream method 4.3.2, 4.3.3, 4.6
getLastModifed method 4.6
getLastModified method 4.3.2
getOutputStream method 4.3.2, 4.3.3, 4.6
getReadTimeout method 4.6
getRequestProperties method 4.6
setAllowUserInteraction method 4.3.2
setConnectTimeout method 4.3.2, 4.6
setDoInput method 4.3.2, 4.6
setDoOutput method 4.3.2, 4.3.3, 4.6
setIfModifiedSince method 4.3.2, 4.6
setReadTimeout method 4.3.2, 4.6
setRequestProperty method 4.3.2, 4.6
setUseCaches method 4.3.2
URLDecoder class 4.6
decode method 4.6
URLEncoder class 4.6
encode method 4.6
URLs 4.3.1
attaching parameters to 4.3.3
connections via 4.3.1
encoding 4.3.3
for databases 5.3.1
for namespace identifiers 3.6
redirecting 4.3.3
relative vs. absolute 3.4.1
URNs 4.3.1
US Letter paper 12.7.1
useLocale method 7.1.4, 7.2.1
User coordinates 12.5.7
User input 11.3.1
User Interface. See Graphical User Interface
User objects 12.4.1
User-Agent request parameter 4.3.3

userNodeForPackage method 10.5
of Preferences 10.5
userRoot method 10.5
of Preferences 10.5
Users
authentication of 9.2
preferences of 2.5
UTC 6.5
UTF-16 1.12, 2.1.4, 2.1.8, 2.2.1
byte order in 2.1.8
in regular expressions 2.7.1
native code and 13.1
UTF-8 2.1.8, 2.2.1
byte order in 2.1.8, 7.7.4
for text files 7.7.1
modified 2.2.1, 7.7.5, 13.1

V

validate method
of Component 11.8.4
validateObject method 2.6.8
Validation 3.4
activating 3.4.1
Value 3.5
value method
of XPathEvaluationResult 3.9
value-of element (XSLT) 3.9
valueChanged method 12.4.5
of TreeSelectionListener 12.7.5
valueForPathChanged method 12.4.6
of TreeModel 12.7.5
ValueLayout interface 13.8.3
withName method 13.8.3
valueOf method (date/time legacy classes) 6.7
VARCHAR 5.2, 5.9.4
varHandle method
of MemoryLayout 13.8.3
Variable 12.4.6
toString method 12.4.6
Variable handle 13.8.1
Variables
binding 8.2.2
fields of 12.4.6
initializing 9.1.5
scope of 8.2.2
Variadic function
foreign 13.8.2
Vendor name, of a reader 12.6.2
Verifiers 9.1.5

Version number, of a reader 12.6.2
Versioning 2.6.7
Views 11.1
visitFile method
of SimpleFileVisitor 2.7.7
visitFile, visitFileFailed methods
of FileVisitor 2.3.8
of SimpleFileVisitor 2.3.8
visitFileFailed method
of SimpleFileVisitor 2.7.7
Visual Basic
event handling in 10.4

W

W3C 3.3, 3.7.1
walk method
of Files 2.3.6
walkFileTree method 2.3.8
of Files 2.7.7
warning method
of ErrorHandler 3.4.1, 3.9
Warnings
SQLWarning 5.4.3
WBMP 12.6.1
WeakReference class 12.4.6
Web applications 5.10
Web crawlers 3.7.1
with SAX parser 3.7.1
with StAX parser 3.7.2
Web pages
dynamic 8.1.6
separating class loaders for 9.1.3
WebRowSet interface 5.7.1
Weekends 6.2
Weeks 7.3
weightx field
of GridBagConstraints 11.8.4
WHERE 5.2
Whitespace
ignoring, while parsing 3.3
in e-mail URIs 4.3.3
in regular expressions 2.7.1
leading/trailing 11.3.1
Wilde, Oscar 7.1
Window class 10.4.7, 10.5
pack method 10.3, 10.5
Window listeners 10.4.4
windowActivated method
of WindowListener 10.5

windowClosed method
of WindowListener 10.5
WindowClosing 11.5.5
windowClosing method
of WindowListener 10.5
windowDeactivated method
of WindowListener 10.5
windowDeiconified method
of WindowListener 10.5
WindowEvent class 10.4.1, 10.4.4, 10.4.7
get*Xxx* methods of 10.4.7
windowFixed method
of Gatherers 1.13
WindowFocusListener interface, methods of 10.4.7
windowIconified method
of WindowListener 10.5
WindowListener interface 10.5
windowActivated method 10.5
windowClosed method 10.5
windowClosing method 10.5
windowDeactivated method 10.5
windowDeiconified method 10.5
windowIconified method 10.5
windowOpened method 10.5
WindowListener interface, methods of 10.4.4, 10.4.7
windowOpened method
of WindowListener 10.5
Windows
activating telnet in 4.1.1
authentication in 9.2.1
character encodings in 7.7
classpath in 5.3.2
glob syntax in 2.3.7
line feed in 2.1.6, 7.7.2
look-and-feel of 12.4.1
paths in 2.1.3, 2.3.1
resources in 7.8
using Microsoft compiler 13.1
Windows operating system
Alt+F4 keyboard shortcut in 11.5.5
pop-up menus in 11.5.4
registry in 10.5
Windows-1252 7.7, 7.7.3
Windows. See Dialogs
windowStateChanged method
of WindowStateListener 10.5
WindowStateListener interface 10.5
windowStateChanged method 10.5
WindowStateListener interface, windowStateChanged method 10.4.7
with keyword 6.3

with method
of LocalDate 6.7
withDayOfMonth method
of LocalDate 6.7
withDayOfYear method
of LocalDate 6.7
withLocale method 6.6, 7.3
of DateTimeFormatter 6.7, 7.9
withMonth method
of LocalDate 6.7
withName method
of ValueLayout 13.8.3
withTargetLayout method
of AddressLayout 13.8.3
with*Xxx* method
of LocalTime 6.7
of Period 6.7
of ZonedDateTime 6.7
withYear method
of LocalDate 6.7
withZoneSameInstant method
of ZonedDateTime 6.7
withZoneSameLocal method
of ZonedDateTime 6.7
Words, in regular expressions 2.7.1
Working directory 2.1.3
wrap method
of ByteBuffer 2.4.2, 2.7.7
WritableByteChannel 2.7.7
write method 2.7.7
WritableByteChannel interface 4.2.4
WritableRaster class 12.6.4, 12.7.5
setDataElements method 12.6.4, 12.7.5
setPixel method 12.7.5
setPixel, setPixels methods 12.6.4
setPixels method 12.7.5
write method
of CipherOutputStream 9.4.4
of Files 2.3.2, 2.7.7
of ImageIO 12.6.1, 12.7.5
of ImageWriter 12.6.3, 12.7.5
of OutputStream 2.1.1, 2.7.7
of WritableByteChannel 2.7.7
of Writer 2.1.2
writeAttribute method 3.8.4
of XMLStreamWriter 3.9
writeBoolean method 2.2.1
of DataOutput 2.7.7
writeByte method 2.2.1
of DataOutput 2.7.7

writeCData method
of XMLStreamWriter 3.9
writeChar method 2.2.1, 2.2.2
of DataOutput 2.7.7
writeCharacters method 3.8.4
of XMLStreamWriter 3.9
writeChars method 2.2.1
of DataOutput 2.7.7
writeComment method
of XMLStreamWriter 3.9
writeDouble method 2.2.1, 2.6.1, 2.6.4
of DataOutput 2.7.7
writeDTD method
of XMLStreamWriter 3.9
writeEmptyElement method 3.8.4
of XMLStreamWriter 3.9
writeEndDocument method
of XMLStreamWriter 3.9
writeEndElement method
of XMLStreamWriter 3.9
writeEndXxx methods (XMLStreamWriter) 3.8.4
writeExternal method 2.6.5
writeFixedString method 2.2.2
writeFloat method 2.2.1
of DataOutput 2.7.7
writeInsert method 12.6.3
of ImageWriter 12.7.5
writeInt method 2.2.1, 2.2.2, 2.6.1
of DataOutput 2.7.7
writeLong method 2.2.1
of DataOutput 2.7.7
writeObject method
of HashSet 2.6.4
of ObjectOutputStream 2.6.1, 2.6.4, 2.7.7
Writer class 2.1, 2.1.2
write method 2.1.2
writeReplace method 2.6.6
writeShort method 2.2.1
of DataOutput 2.7.7
writeStartDocument method
of XMLStreamWriter 3.9
writeStartElement method
of XMLStreamWriter 3.9
writeStartXxx methods (XMLStreamWriter) 3.8.4
writeString method 2.3.2
of Files 2.7.7
writeUTF method 2.2.1
of DataOutput 2.7.7

X

X.509 format 9.3.4
XHTML 3.1, 3.7.1
XML 3, 3.9
annotated version of the standard 3.1
attributes in 3.1
case sensitivity of 3.1
end and empty tags in 3.1
in databases 5.9.4
namespaces in 3.6
vs. HTML 3.1
XML catalogs 3.4.1
XML documents
DTDs in 3.2, 3.4.1
format of 3.1
generating 3.8
from non-XML legacy data 3.9
HTML files from 3.9
plain text from 3.9
with StAX 3.8.4
locating information in 3.5
malformed 3.8.4
parsing 3.3
structure of 3.2, 3.3, 3.4
validating 3.4
with/without namespaces 3.8.1, 3.8.2
XML Schema 3.4, 3.4.2
attributes in 3.4.2
documentation for 3.6
parsing with 3.4.2
referencing in XML documents 3.4.2
repeated elements in 3.4.2
type definitions in 3.4.2
XMLInputFactory class 3.9
createXMLStreamReader method 3.9
newInstance method 3.9
setProperty method 3.7.2, 3.9
Xmlns 3.6
XMLOutputFactory class 3.9
createXMLStreamWriter method 3.8.4, 3.9
newInstance method 3.8.4, 3.9
XMLReader interface 3.9
implementing 3.9
parse method 3.9
setContentHandler method 3.9
XMLStreamReader interface 3.9
getAttributeCount method 3.9
getAttributeLocalName method 3.9
getAttributeName method 3.9
getAttributeValue method 3.9
getAttributeXxx methods 3.7.2

getLocalName method 3.9
getName method 3.9
getText method 3.9
hasNext method 3.9
isCharacters method 3.9
isEndElement method 3.9
isStartElement method 3.9
isWhiteSpace method 3.9
next method 3.9
XMLStreamWriter interface 3.8.4, 3.9
close method 3.9
not autocloseable 3.8.4
setDefaultNamespace method 3.9
setPrefix method 3.9
writeAttribute method 3.8.4, 3.9
writeCData method 3.9
writeCharacters method 3.8.4, 3.9
writeComment method 3.9
writeDTD method 3.9
writeEmptyElement method 3.8.4, 3.9
writeEndDocument method 3.9
writeEndElement method 3.9
writeEndXxx methods 3.8.4
writeStartDocument method 3.9
writeStartElement method 3.9
writeStartXxx methods 3.8.4
XOR composition rule 12.5.9
XPath interface 3.5, 3.9
count function 3.5
evaluate method 3.5, 3.9
evaluateExpression method 3.5, 3.9
XPathEvaluationResult interface 3.9
type method 3.9
value method 3.5, 3.9
XPathFactory class 3.9
newInstance method 3.5, 3.9
newXpath method 3.9
XPathNodes interface 3.5
xs:, xsd: prefixes (XML Schema) 3.4.2
Xsd:attribute 3.4.2
Xsd:choice 3.4.2
Xsd:complexType 3.4.2
Xsd:element 3.4.2
Xsd:enumeration 3.4.2
Xsd:schema 3.4.2
Xsd:sequence 3.4.2
Xsd:simpleType 3.4.2
Xsl:apply-templates 3.9
Xsl:output 3.9
Xsl:template 3.9

Xsl:value-of 3.9
XSLT 3.8.3, 3.9
copying attribute values in 3.9
templates in 3.9
XSLT processors 3.9

Y

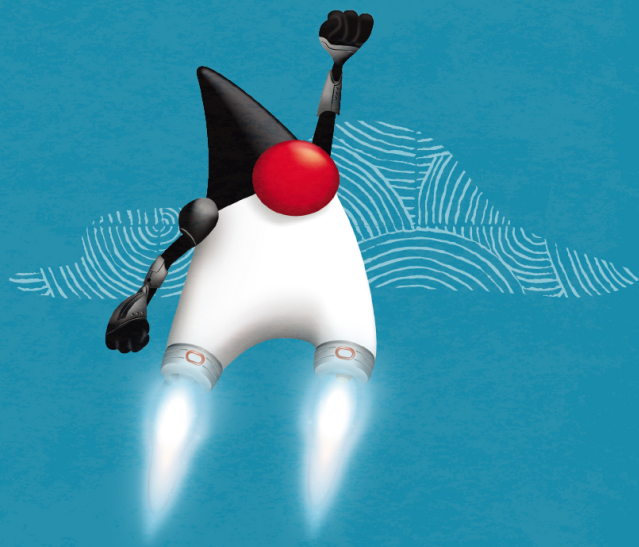
Year, YearMonth classes 6.2

Z

Z (boolean), type code 2.6.2
ZIP archives 2.2.3
reading 2.1.3, 2.2.3
writing 2.2.3
Zip code lookup 4.3.3
ZIP file systems 2.3.9
ZipEntry class 2.7.7
Constructor 2.7.7
getCrc method 2.7.7
getName method 2.7.7
getSize method 2.7.7
isDirectory method 2.7.7
setCrc method 2.7.7
setMethod method 2.7.7
setSize method 2.7.7
ZipInputStream class 2.1.2, 2.2.3, 2.7.7
closeEntry method 2.2.3, 2.7.7
Constructor 2.7.7
getNextEntry method 2.2.3, 2.7.7
read method 2.2.3
ZipOutputStream class 2.1.2, 2.2.3, 2.7.7
closeEntry method 2.2.3, 2.7.7
Constructor 2.7.7
putNextEntry method 2.2.3, 2.7.7
setLevel method 2.7.7
setMethod method 2.7.7
ZonedDateTime class 6.5, 6.7, 7.9
format method 7.3
from method 6.7
getDayOfMonth method 6.7
getDayOfWeek method 6.7
getDayOfYear method 6.7
getHour method 6.7
getMinute method 6.7
getMonth method 6.7
getMonthValue method 6.7
getNano method 6.7
getOffset method 6.7

getSecond method 6.7
getYear method 6.7
isAfter method 6.7
isBefore method 6.7
legacy classes and 6.7
minus method 6.7
minus*Xxx* method 6.7
now method 6.7
of method 6.7
of, ofInstant methods 6.5
ofInstant method 6.7
parse method 6.7, 7.3, 7.9
plus method 6.7
plus*Xxx* method 6.7
toInstant method 6.5, 6.7
toLocalDate method 6.7
toLocalDateTime method 6.7
toLocalTime method 6.7
with*Xxx* method 6.7
withZoneSameInstant method 6.7
withZoneSameLocal method 6.7
ZoneId class 6.7
getAvailableZoneIds method 6.5
of method 6.5

Java™



The #1 language

for today's tech trends

www.oracle.com/java

ORACLE
CLOUD
Infrastructure

Build, test, and deploy applications on OCI—for free

The next-generation cloud designed to run any application, faster and more securely, for less.

Learn more about why customers are choosing OCI and try it for free.
oracle.com/cloud/free



ORACLE
CLOUD
Infrastructure

Learn.
Build.
Launch.



(And don't forget to BYOOST*)

Consider this an open invite to join developers like you to pick up new skills—and pick your favorite language—to build whatever you want on OCI.

Check us out and meet new friends at developer.oracle.com.

*Bring your own open source tools. We'll bring the fun-sized ideas.

ORACLE

Get answers to your technical questions

Find getting started guides, documentation, tutorials, architectures, and more at Oracle Help Center – Oracle's #1 hub for technical documentation.







ORACLE

Goodbye, database complexity. Hello, simplicity.

Reduce complexity and cost with the one database for all your workloads. Cloud native, converged, and automated, Oracle Autonomous Database eliminates manual maintenance hassles and delivers top performance, security, and automatic scaling for your changing needs. It also supports the full development lifecycle for modern applications.

Get simplicity in the cloud or on-premises.

Oracle Autonomous Database
Learn more at
oracle.com/autonomous-database



ORACLE

Beta Test Oracle Software

Get a first look at Oracle's latest products, features, and cloud services – and help perfect them! Work closely with Oracle's product development teams to share your feedback and suggestions for product improvements.

To view and apply for beta program participation, check out the Oracle Beta Testing Opportunities site:

https://pdpm.oracle.com/pls/apex/f?p=102:250



Register Your Product at informit.com/register

Access additional benefits and save up to 65%* on your next purchase

- Automatically receive a coupon for 35% off books, eBooks, and web editions and 65% off video courses, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.**
- Check the box to hear from us and receive exclusive offers on new editions and related products.

InformIT is the trusted technology learning source and online home of information technology brands at Pearson, the world's leading learning company. At informit.com you can shop our books, eBooks, and video training. Most eBooks are DRM-free and include PDF and EPUB files.

- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletter (informit.com/newsletters).
- Access thousands of free chapters and video lessons.
- Enjoy free ground shipping on U.S. orders.*

** Offers subject to change.*
*** Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

Connect with InformIT—Visit informit.com/community

Pearson

informIT

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Oracle Press • Peachpit Press • Pearson IT Certification